Functional Design Patterns

@ScottWlaschin

fsharpforfunandprofit.com

fsharpWorks

Practices?

Approaches?

Functional Design ~~Patterns~~

Tips?

@ScottWlaschin

fsharpforfunandprofit.com

fsharpWorks

Functional Design Patterns

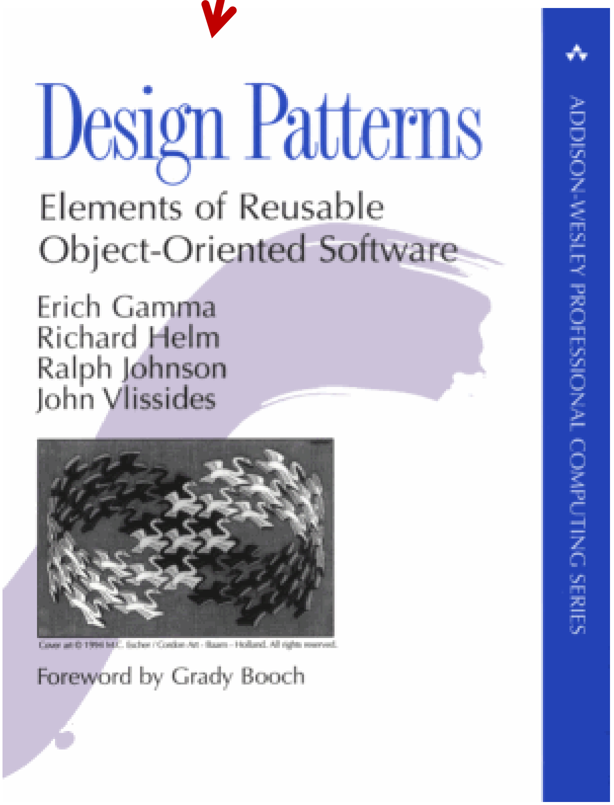
So I'll reluctantly stick with this...

@ScottWlaschin

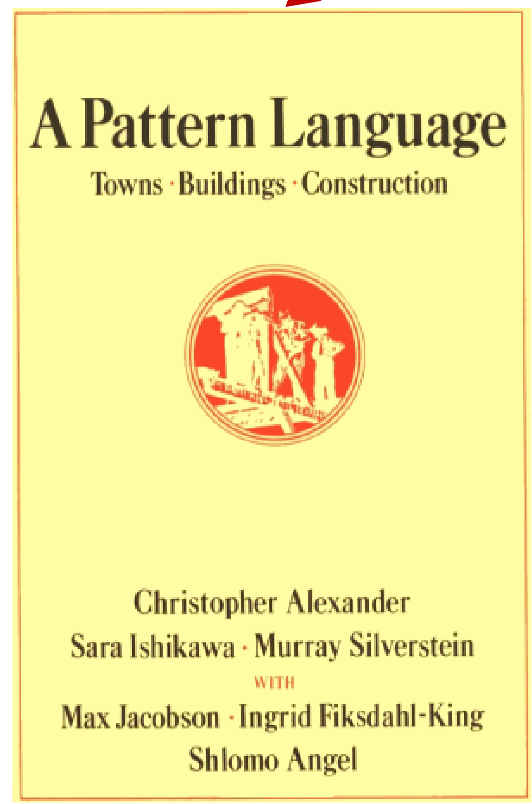
fsharpforfunandprofit.com

fsharpWorks

Not this kind of pattern



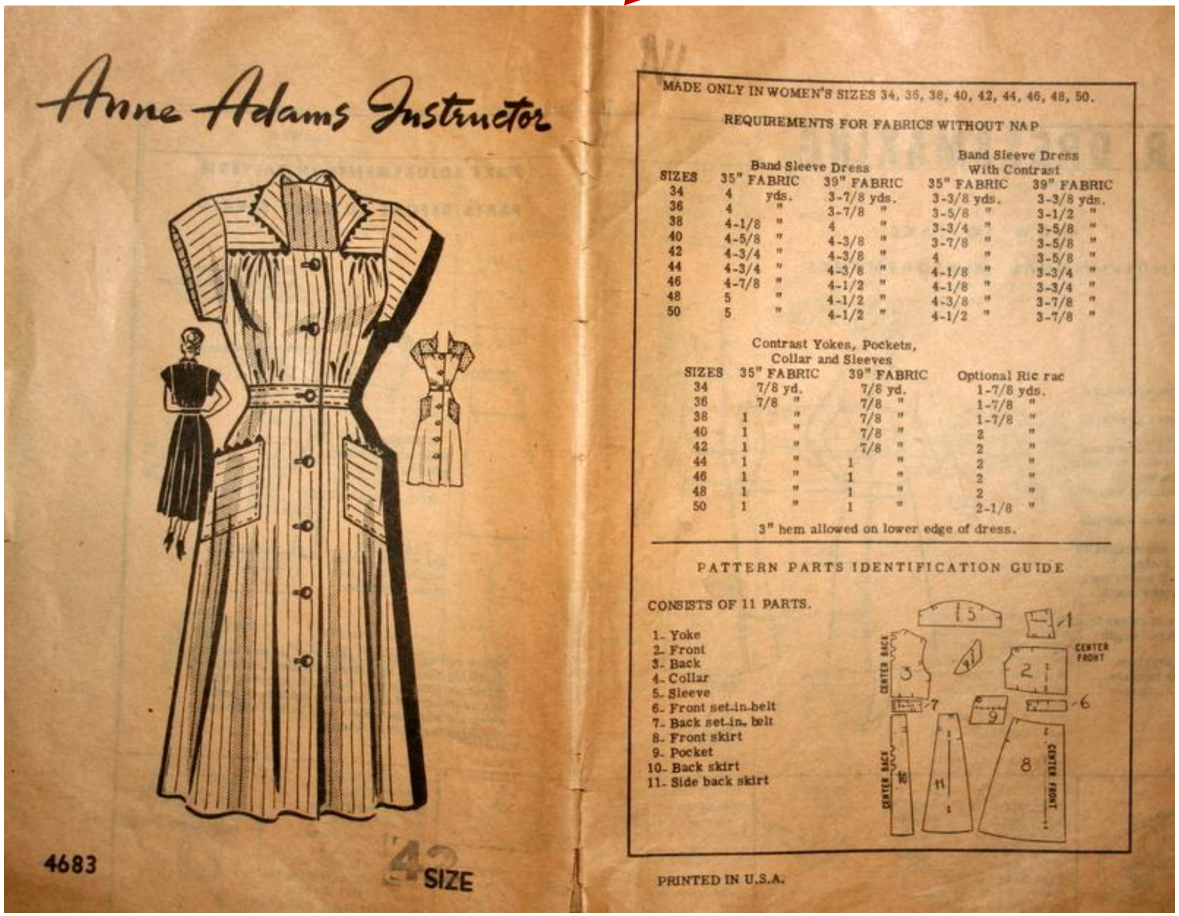
nor this



These kind of patterns

All about recognizing repetition.

This kind of pattern!



A mixture of guidelines, templates, & construction advice

Functional patterns

- Apomorphisms
- Dynamorphisms
- Chronomorphisms
- Zygohistomorphic prepromorphisms

Functional patterns

- Core Principles of FP design
  - Functions, types, composition
- Functions as parameters
  - Abstraction, Dependency injection
  - Partial application, Continuations, Folds,
- Chaining functions
  - Error handling, Async
  - Monads
- Dealing with wrapped data
  - Lifting, Functors
  - Validation with Applicatives
- Aggregating data and operations
  - Monoids

This talk

Functional programming is scary

Functor

Currying

Catamorphism

Applicative

Monad

Monoid

Functional programming is ~~scary~~ unfamiliar

“Mappable”

Currying

“Collapsable”

Applicative

“Chainable”

“Aggregatable”

This!

Object oriented programming is scary

Generics

Interface

Polymorphism

Inheritance

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

SOLID

Covariance

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

OO pattern/principle	FP pattern/principle
Single Responsibility Principle	Functions
Open/Closed principle	Functions
Dependency Inversion Principle	Functions, also
Interface Segregation Principle	Functions
Factory pattern	Yes, functions
Strategy pattern	Oh my, functions again!
Decorator pattern	Functions
Visitor pattern	Functions ☺

Seriously, FP patterns are different

A short and mostly wrong history of programming paradigms

Object-Oriented programming

- What we envisioned:
  - Smalltalk
- What we got:
  - C++
  - Object oriented Cobol
  - PHP
  - Java

“Worse is better”

Functional programming

- What we envisioned:
  - Haskell, OCaml, F#, etc
- What we got:
  - ??

Please don't let this happen to FP!

Important to understand!

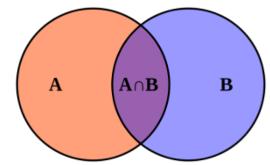
CORE PRINCIPLES OF FP DESIGN

Core principles of FP design

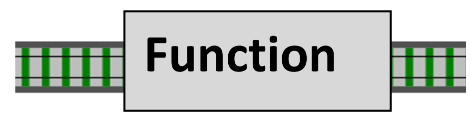
Steal from mathematics

$$\begin{tikzcd} T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"'] & T \end{tikzcd} \qquad \begin{tikzcd} T \arrow[r, "\eta T"] \arrow[d, "T\eta"'] \arrow[dr, equal] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"'] & T \end{tikzcd}$$

Types are not classes



Functions are things



Composition everywhere

Core principle:
Steal from mathematics

"Programming is one of the most difficult branches of applied mathematics"
- E. W. Dijkstra

$$
\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}
\qquad
\begin{tikzcd}
T \arrow[r, "\eta T"] \arrow[d, "T\eta"'] \arrow[dr, equal] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}
$$

Why mathematics

Dijkstra said:

- Mathematical assertions tend to be unusually **precise**.
- Mathematical assertions tend to be **general.** They are applicable to a large (often infinite) class of instances.
- Mathematics embodies a discipline of reasoning allowing assertions to be made with an **unusually high confidence level**.

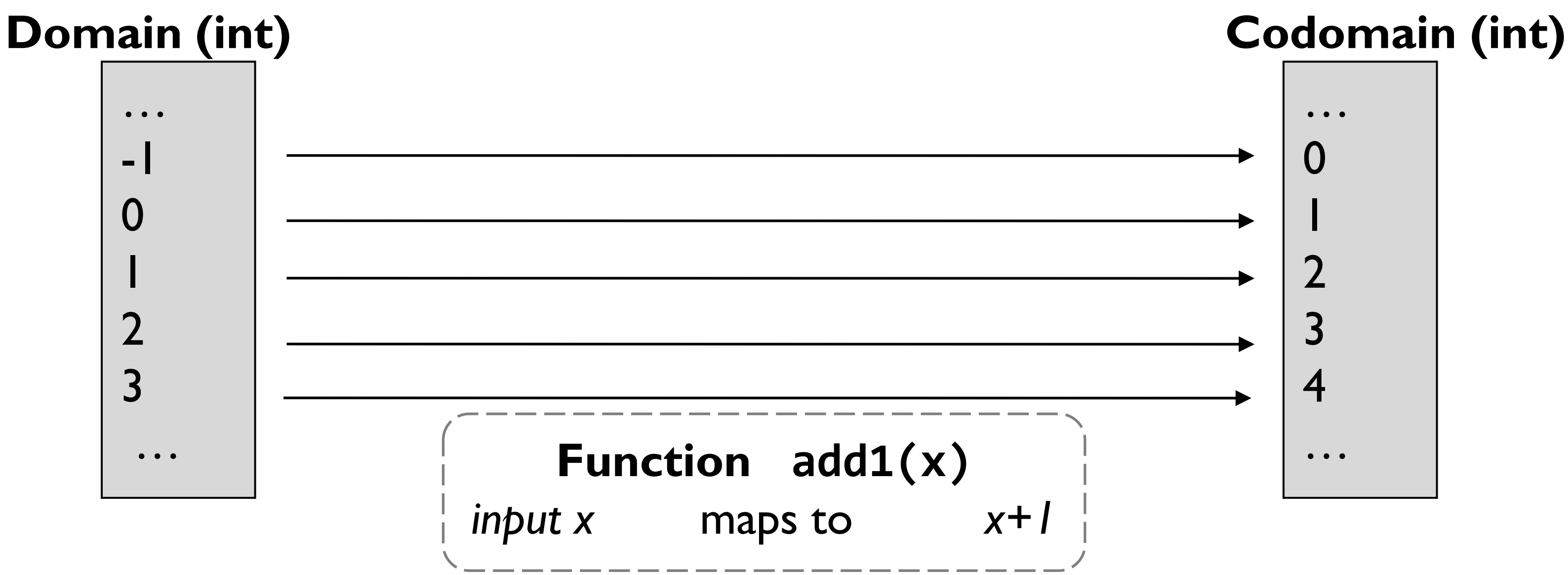
"Object-oriented programming is an exceptionally bad idea which could only have originated in California."

E. W. Dijkstra

"You probably know that arrogance, in computer science, is measured in nanodijkstras."

- Alan Kay

Mathematical functions




```
let add1 x = x + 1

val add1 : int -> int
```

Mathematical functions

A function doesn't "do" anything

Domain (int)

...
-1
0
1
2
3
...

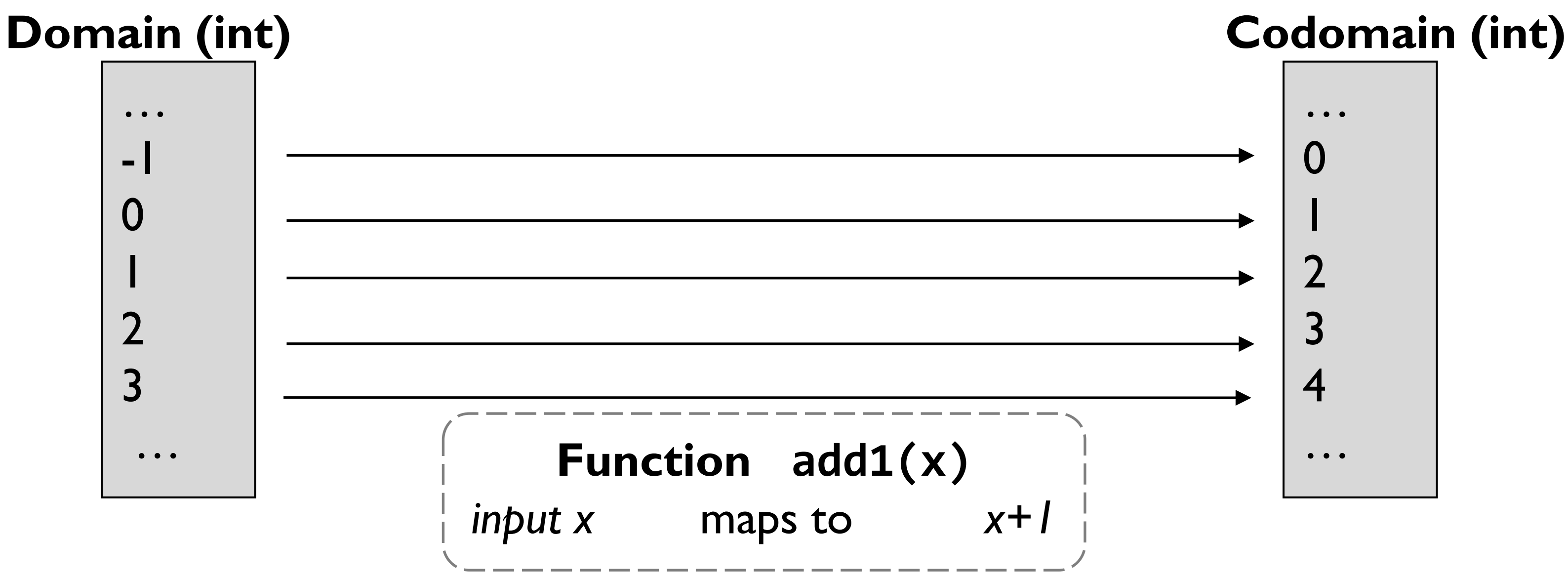
```
int add1(int input)
{
    switch (input)
    {
    case 0: return 1;
    case 1: return 2;
    case 2: return 3;
    case 3: return 4;
    etc ad infinitum
    }
}
```

Codomain (int)

...
0
1
2
3
4
...

- Input and output values already exist
- A function is not a calculation, just a mapping
- Input and output values are unchanged (immutable)

Mathematical functions



- A (mathematical) function always gives the same output value for a given input value
- A (mathematical) function has no side effects

Functions don't have to be about arithmetic

Customer (domain)

…
Cust1
Cust2
Cust3
Cust3
…

```
Name CustomerName(Customer input)
{
   switch (input)
   {
   case Cust1: return "Alice";
   case Cust2: return "Bob";
   case Cust3: return "Sue";
   etc
   }
}
```

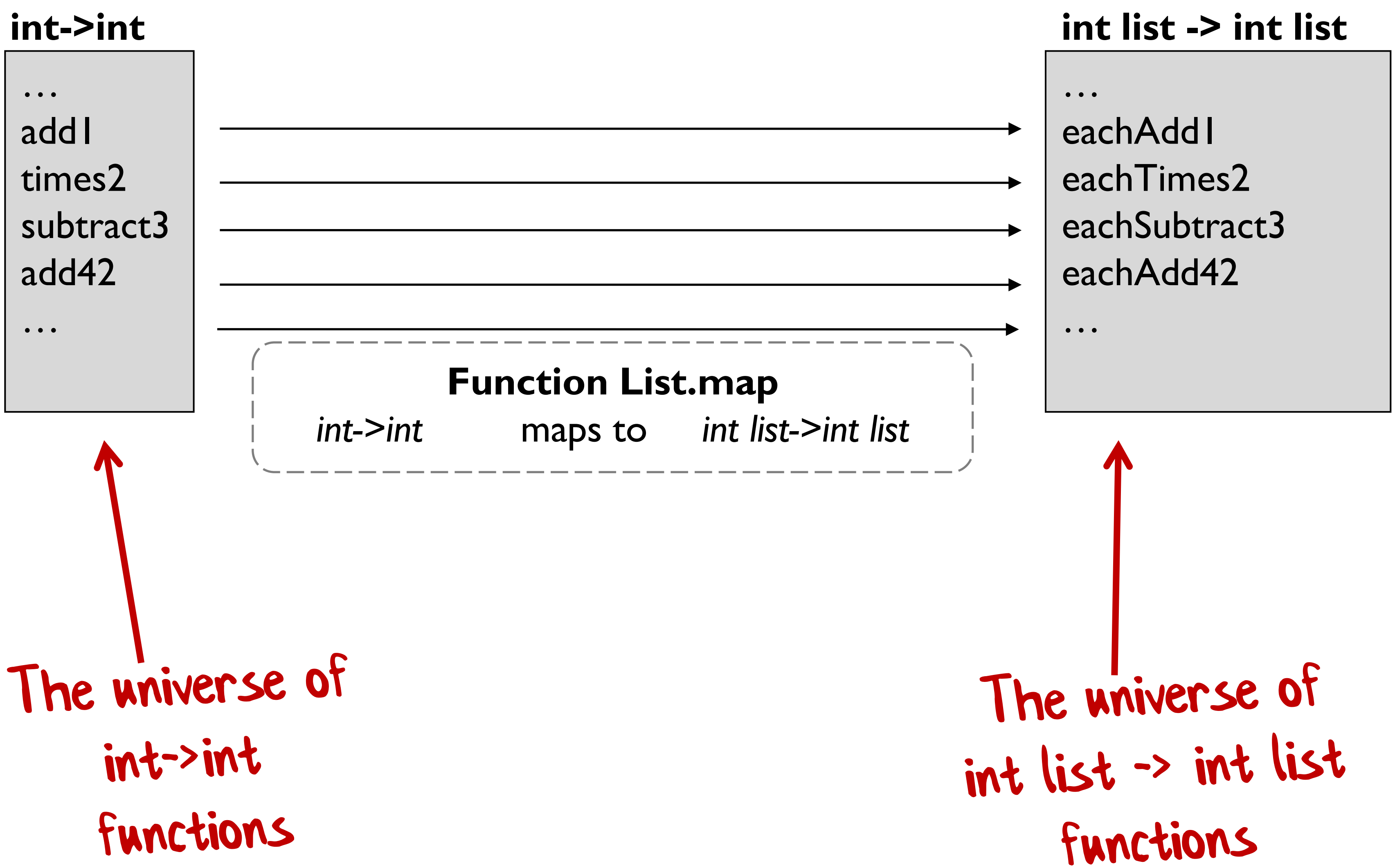
Name (codomain)

…
Alice
Bob
Sue
John
Pam
…

The universe of Customers

The universe of Personal Names

Functions can work on functions



Guideline:

Strive for purity

not always achievable
or desirable

Pure functions are easy to reason about

Reasoning about code that might not be pure:

```
customer.SetName(newName);
```

```
var name = customer.GetName();
```

Is the customer being changed?

Reasoning about code that is pure:

```
let newCustomer = setCustomerName(aCustomer, newName)
```

```
let name,newCustomer = getCustomerName(aCustomer)
```

The customer is being changed.

??? I can tell something is funny just by looking at the code

"Don't trust the name – trust the signature"

Pure functions are easy to refactor

```
let x = doSomething()
let y = doSomethingElse(x)
return y + 1
```

Pure functions are easy to refactor

```
let x = doSomething()
let y = doSomethingElse(x)
return y + 1
```

This can be extracted into a new function with cut/paste!

Pure functions are easy to refactor

```
let helper() =
    let x = doSomething()
    let y = doSomethingElse(x)
    return y

return helper() + 1
```

No Resharper needed!

More practical benefits of pure functions

- Laziness
  - only evaluate when I need the output
- Cacheable results
  - same answer every time
  - "memoization"
- No order dependencies
  - I can evaluate them in any order I like
- Parallelizable
  - "embarrassingly parallel"

How to design a pure function

No Side Effects

Immutable Data

Pure Awesomeness!

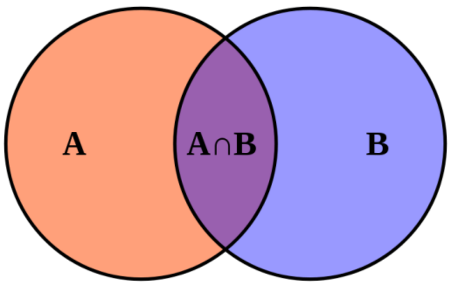
Pun intended

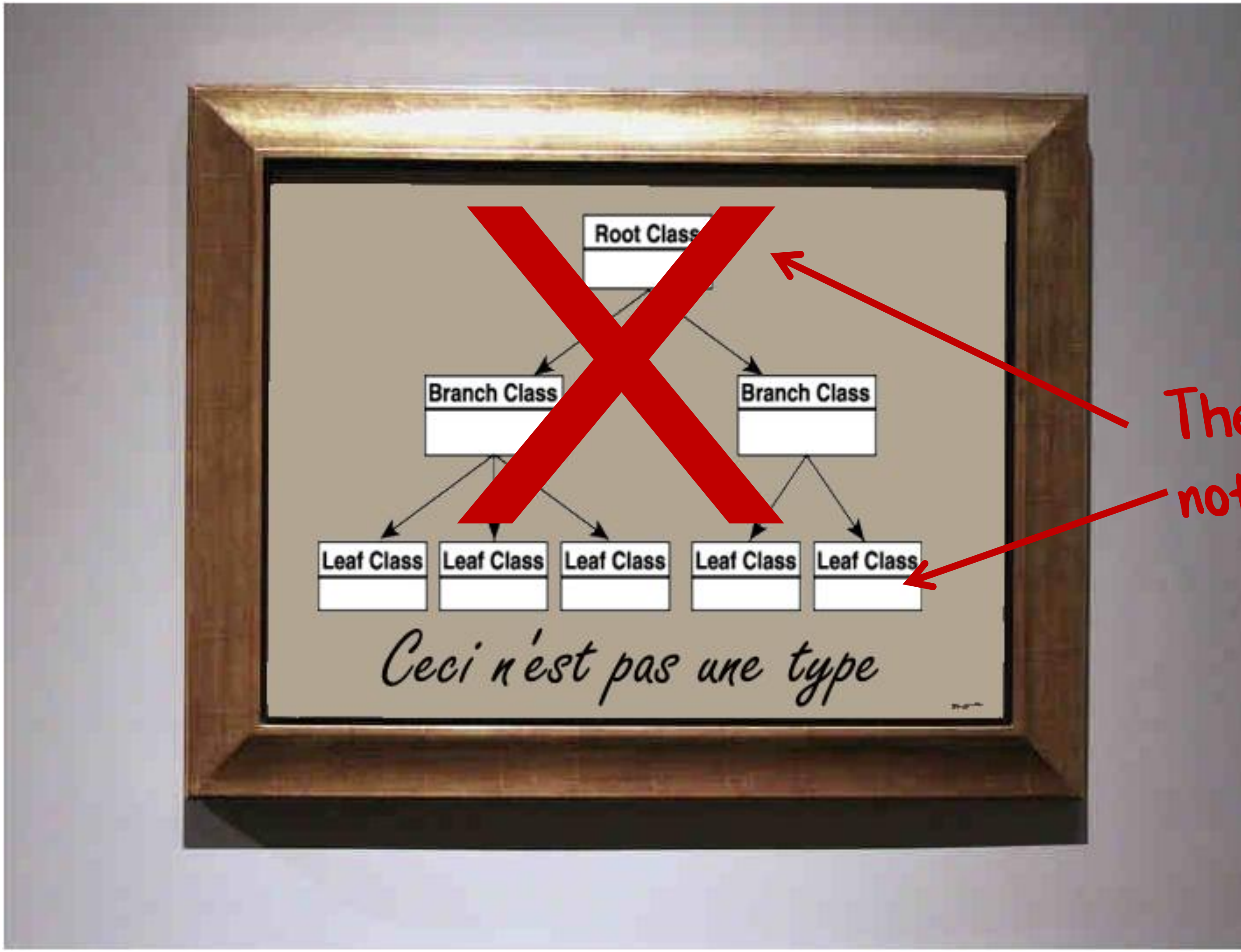
How to design a pure function

The language can help a lot!

- Haskell
  - very pure
- F#, OCaml, Clojure, Scala
  - easy to be pure
- C#, Java, JavaScript
  - have to make an effort

Core principle:
Types are not classes

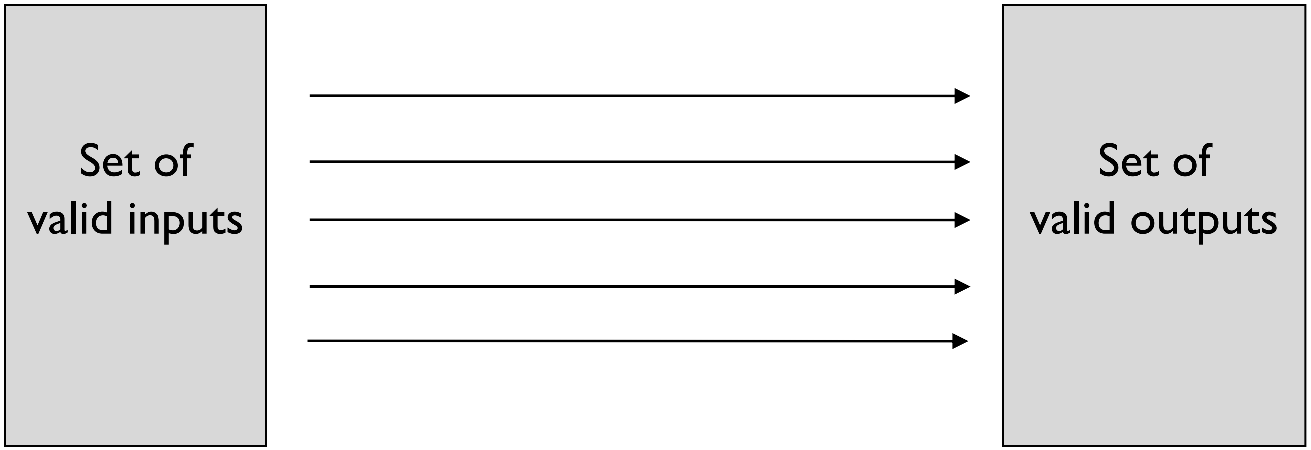




These are not types

Types are not classes

So what is a type?



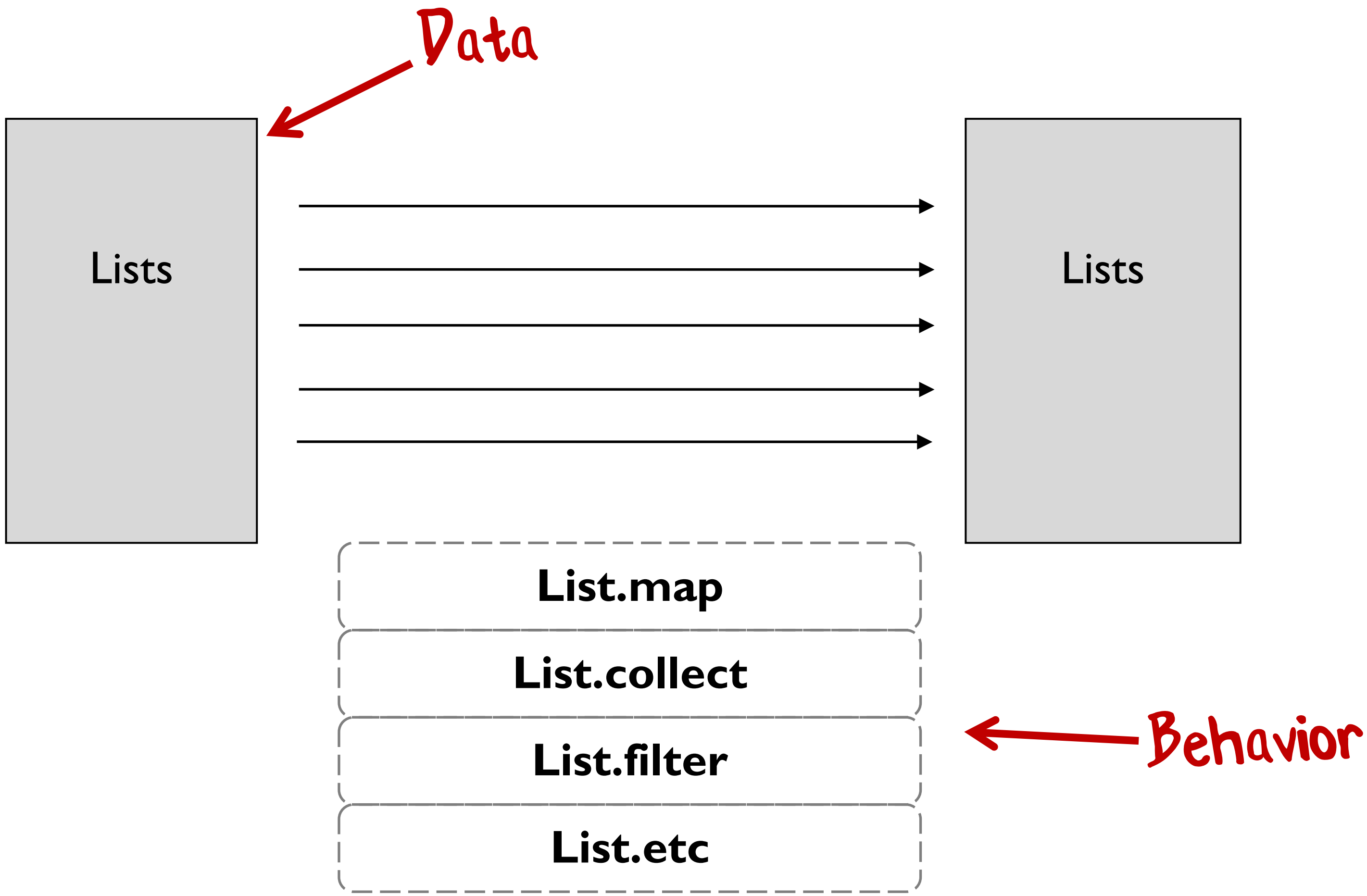
These are types

"Int" is a type

"Customer" is a type

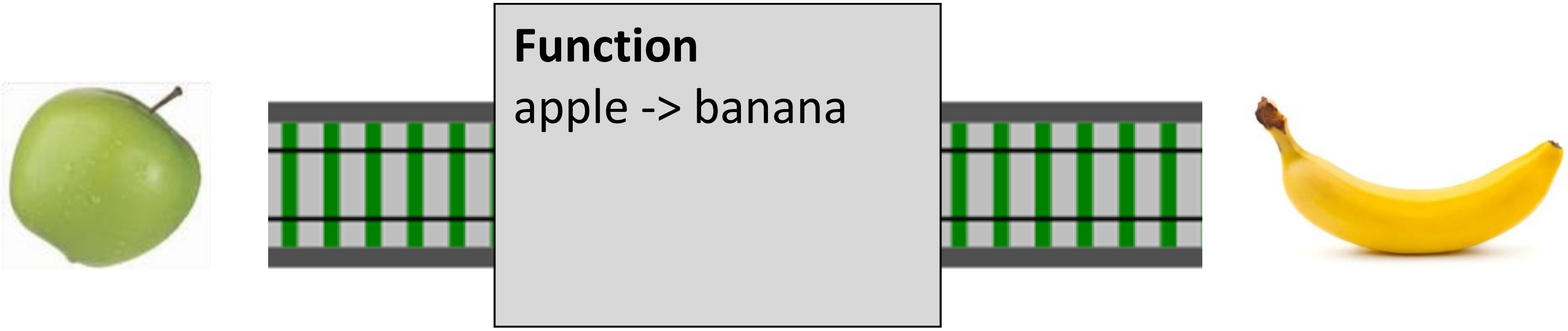
"int->int" is a type

Types separate data from behavior



Core principle:
Functions are things

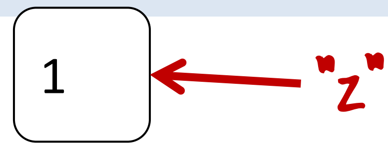
Functions as things



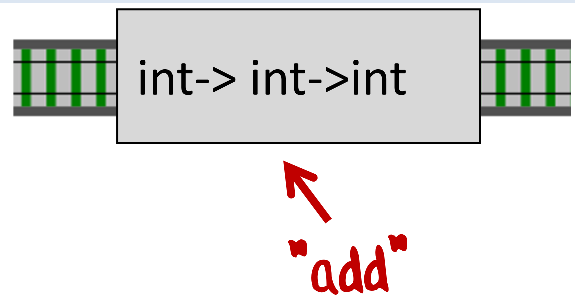
A function is a standalone thing,
not attached to a class

Functions as things

```
let z = 1
```



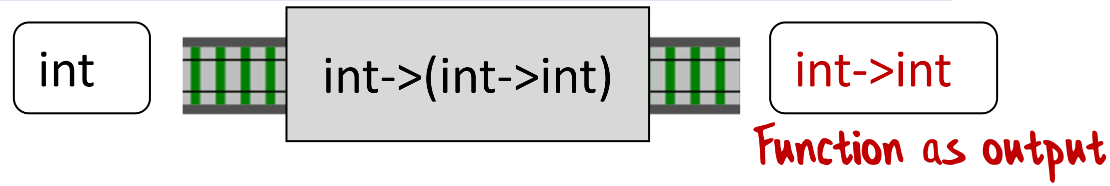

```
let add x y = x + y
```



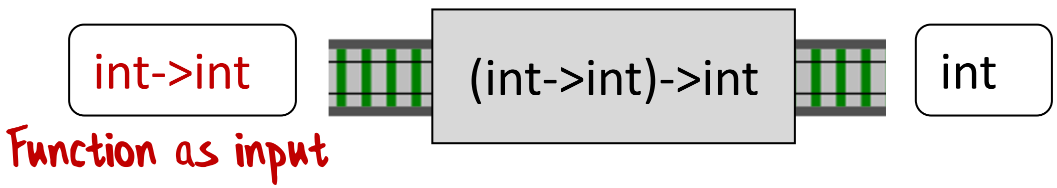
Same keyword
(not a coincidence)

Functions as inputs and outputs

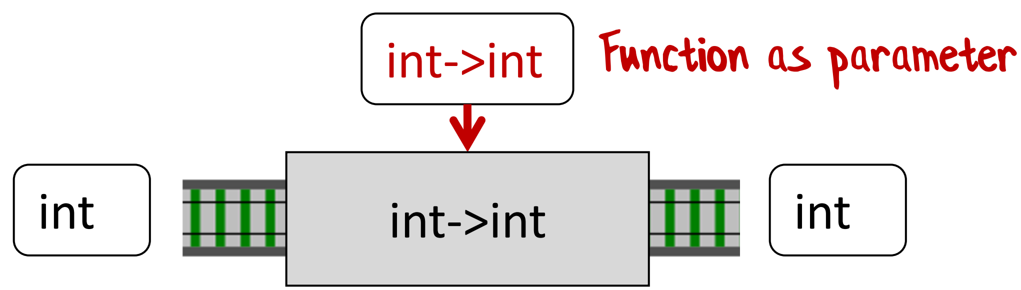
```
let add x = (fun y -> x + y)
```




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



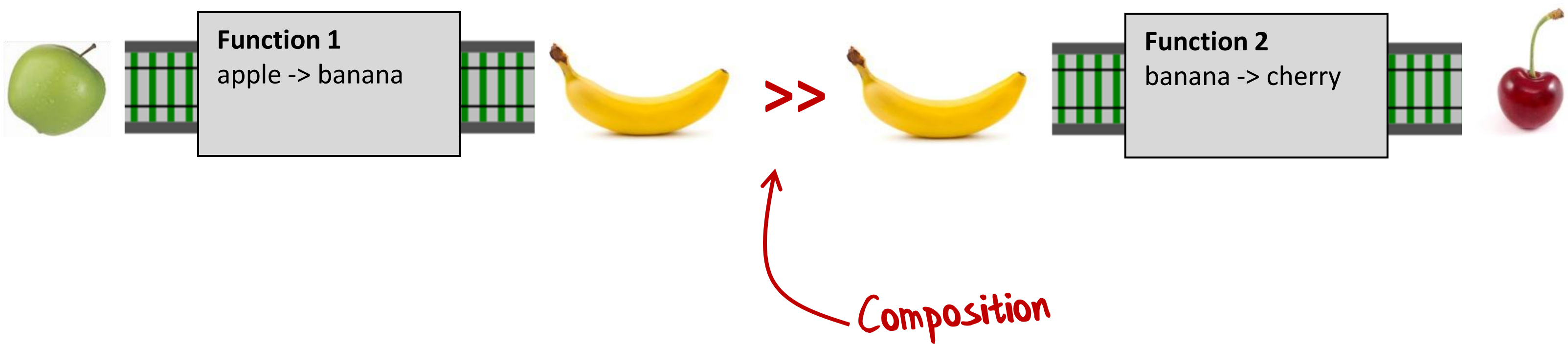

```
let transformInt f x = (f x) + 1
```



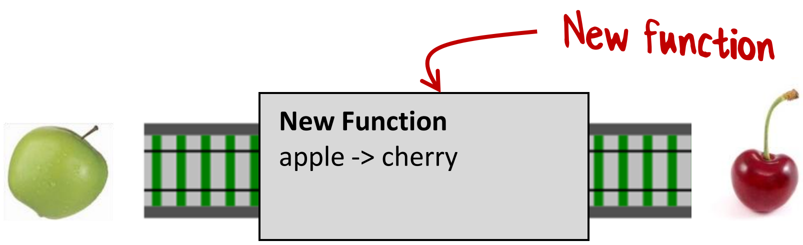
Core principle:
Composition everywhere

Function composition

Function composition



Function composition



Can't tell it was built from smaller functions!

Types can be composed too

"algebraic types"

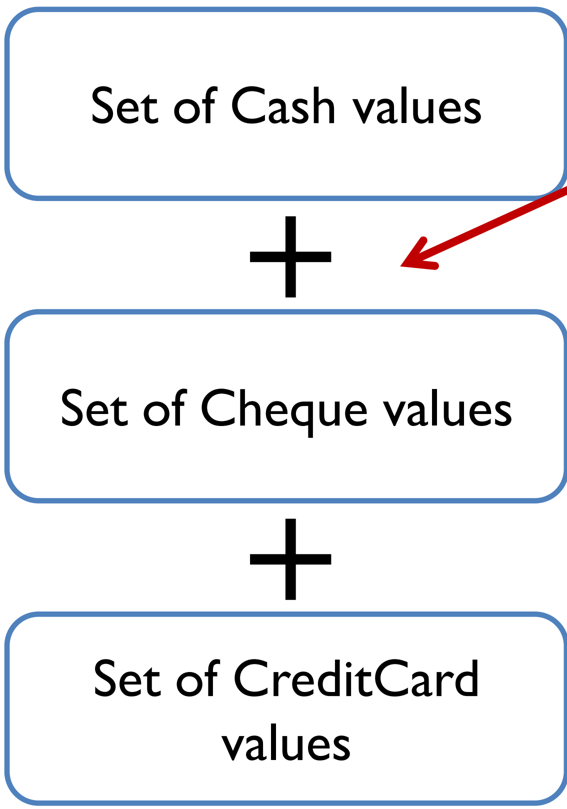
Product types

Set of people × Set of dates = Alice, Jan 12th
Bob, Feb 2nd
Carol, Mar 3rd

"product type"

type **Birthday** = Person * Date

Sum types



"sum type"

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

DDD & DOMAIN MODELLING

Domain modelling pattern:

Use types to represent constraints

Avoid "primitive obsession"

Types represent constraints on input and output

type **Suit** = Club | Diamond | Spade | Heart

type **String50** = // non-null, not more than 50 chars

type **EmailAddress** = // non-null, must contain '@'

type **StringTransformer** = string -> string

Instant mockability

type **GetCustomer** = CustomerId -> Customer option

Domain modelling pattern:
Types are cheap

"There's no problem that can't be solved by wrapping it in a type"

Types are cheap

type **Suit** = Club | Diamond | Spade | Heart

type **Rank** = Two | Three | Four | Five | Six | Seven | Eight
| Nine | Ten | Jack | Queen | King | Ace

New types composed from existing types

type **Card** = Suit * Rank

type **Hand** = Card list

Only one line of code to create a type!

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
...

What happens here?

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
...

0 is missing

0 is 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 types to indicate errors

Output types as error codes

ParseInt: string -> int

ParseInt: string -> int option

No nulls
No exceptions

FetchPage: Uri -> String

FetchPage: Uri -> SuccessOrFailure<String>

LoadCustomer: CustomerId -> Customer

LoadCustomer: CustomerId -> SuccessOrFailure<Customer>

Use the signature, Luke!

Domain modelling principle:

"Make illegal states unrepresentable"

Types can represent business rules

```
type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmailAddress

type ContactInfo =
    | EmailOnly of EmailContactInfo
    | AddrOnly of PostalContactInfo
    | EmailAndAddr of EmailContactInfo * PostalContactInfo
```

Domain modelling principle:
Use sum-types instead of inheritance

Using sum vs. inheritance

“closed” set of options

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

extra data is obvious

OO version:

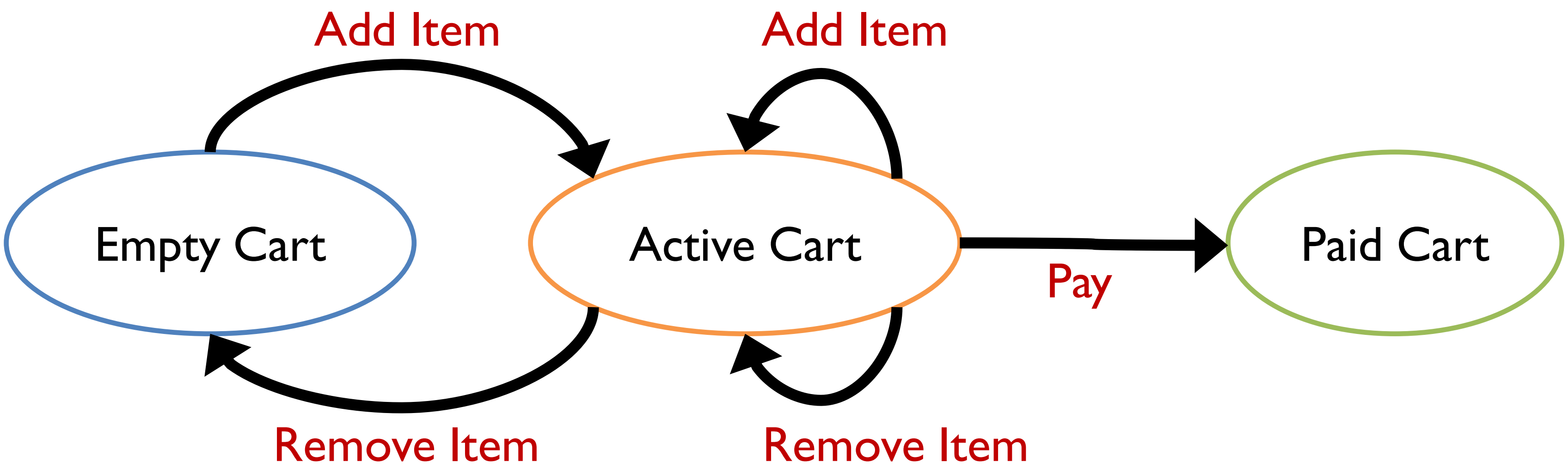
```
interface IPaymentMethod {..}
class Cash : IPaymentMethod {..}
class Cheque : IPaymentMethod {..}
class Card : IPaymentMethod {..}
class Evil : IPaymentMethod {..}
```

What goes in here? What is the common behaviour?

Definition is scattered around many locations

“open” set of options – unpleasant surprises?

Domain modelling principle:
Use sum-types for state machines




```
type ShoppingCart =
    | EmptyCartState
    | ActiveCartState of ActiveCartData
    | PaidCartState of PaidCartData
```

Domain modelling principle:
It's ok to expose public data

It's ok to expose public data

Immutable

```
type PersonalName = {
    FirstName: String50
    MiddleInitial: String1 option
    LastName: String50 }
```

Can't create invalid values

Domain modelling principle:
Types are executable documentation

Types are executable documentation

```
type Suit = Club | Diamond | Spade | Heart
type Rank = Two | Three | Four | Five | Six | Seven | Eight
            | Nine | Ten | Jack | Queen | King | Ace
type Card = Suit * Rank

type Hand = Card list
type Deck = Card list

type Player = {Name:string; Hand:Hand}
type Game = {Deck:Deck; Players: Player list}

type Deal = Deck –› (Deck * Card)
type PickupCard = (Hand * Card) –› Hand
```

Types can be nouns

Types can be verbs

Types are executable documentation

```
type CardType = Visa | Mastercard
type CardNumber = CardNumber of string
type ChequeNumber = ChequeNumber of int

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

Can you guess what payment methods are accepted?

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

Static types only!
Sorry Clojure and JS developers ☹

HIGH-LEVEL DESIGN

Design paradigm:
Functions all the way down

~~“Functions in the small,~~
~~objects in the large”~~

“Functions in the small,
functions in the large”

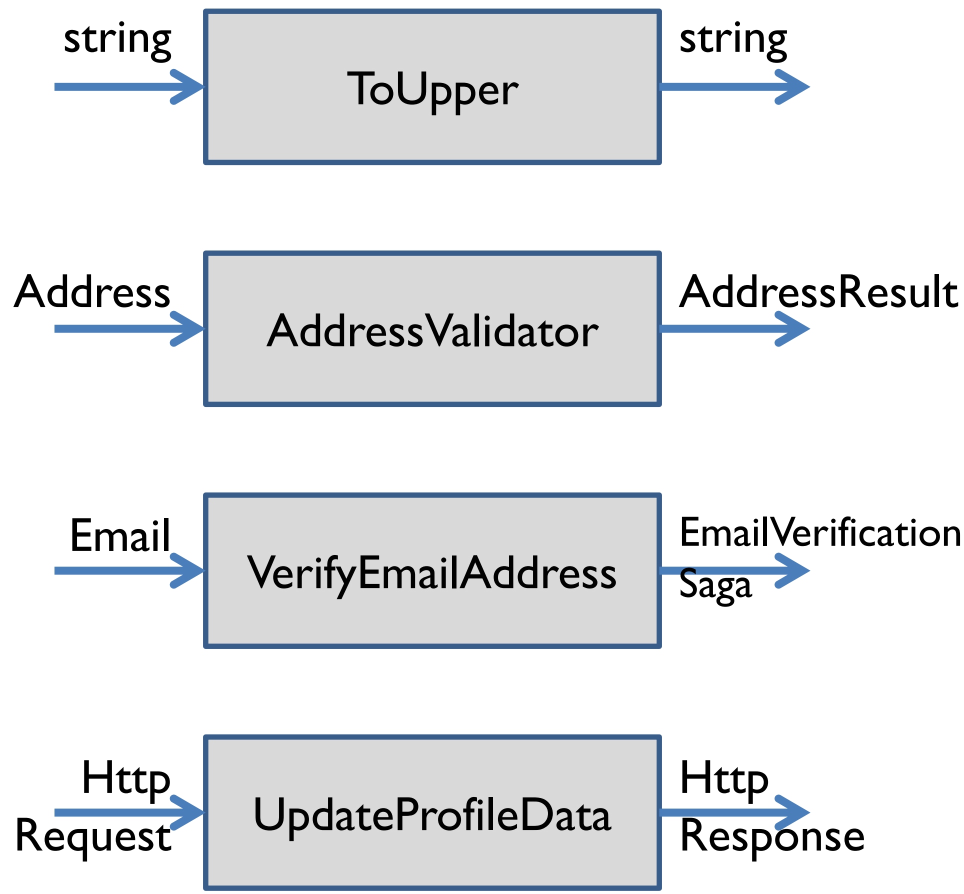
Low-level operation

Service

“Service” is the new “microservice”

Domain logic

High-level use-case



“Composition is fractal”

Design paradigm:
Transformation-oriented programming

Interacting with the outside world

```
type ContactDTO = {
  FirstName: string
  MiddleInitial: string
  LastName: string

  EmailAddress: string
  IsEmailVerified: bool
  }
```

External model

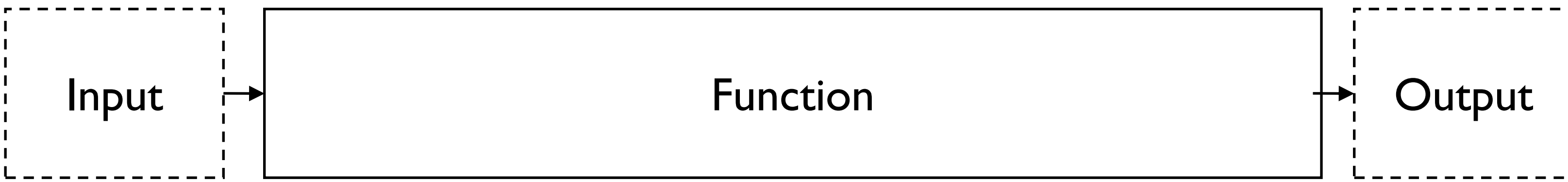
Transformation to clean internal model

```
type EmailAddress = ...
```

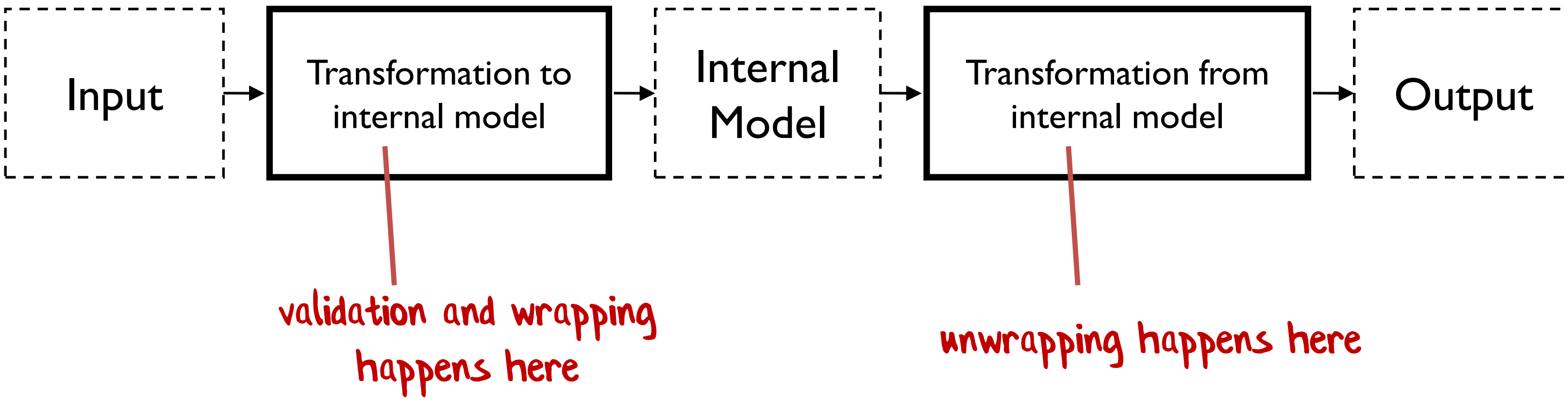
Internal model

```
type VerifiedEmail =
  VerifiedEmail of EmailAddress

type EmailContactInfo =
  | Unverified of EmailAddress
  | Verified of VerifiedEmail

type PersonalName = {
  FirstName: String50
  MiddleInitial: String1 option
  LastName: String50 }

type Contact = {
  Name: PersonalName
  Email: EmailContactInfo }
```

Transformation oriented programming

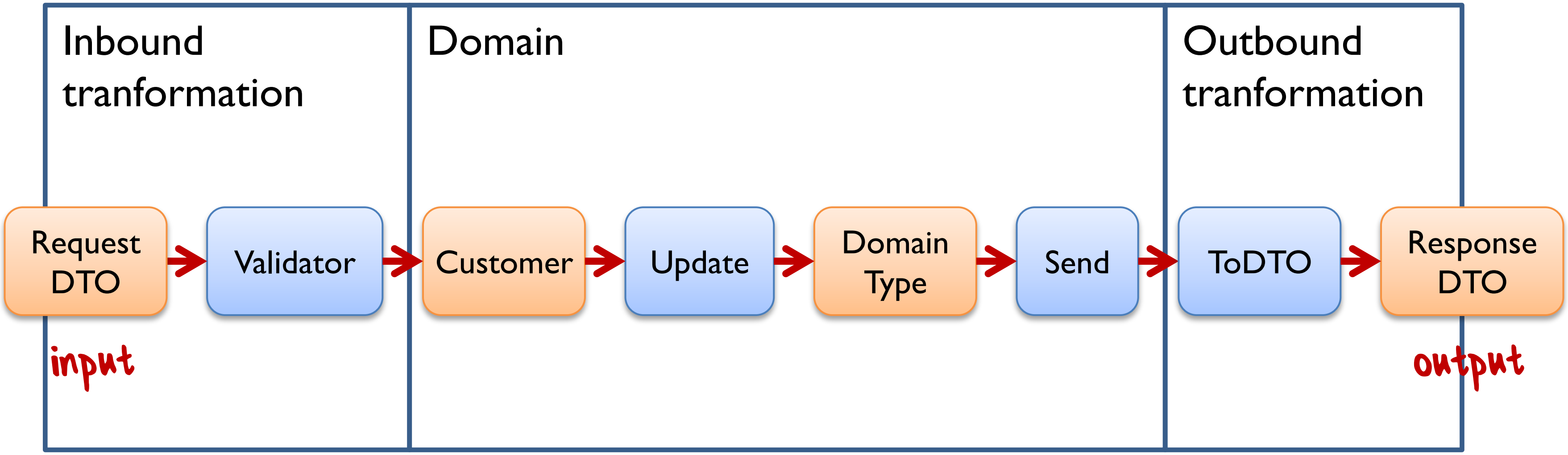


Transformation oriented programming



"transformation-oriented programming"

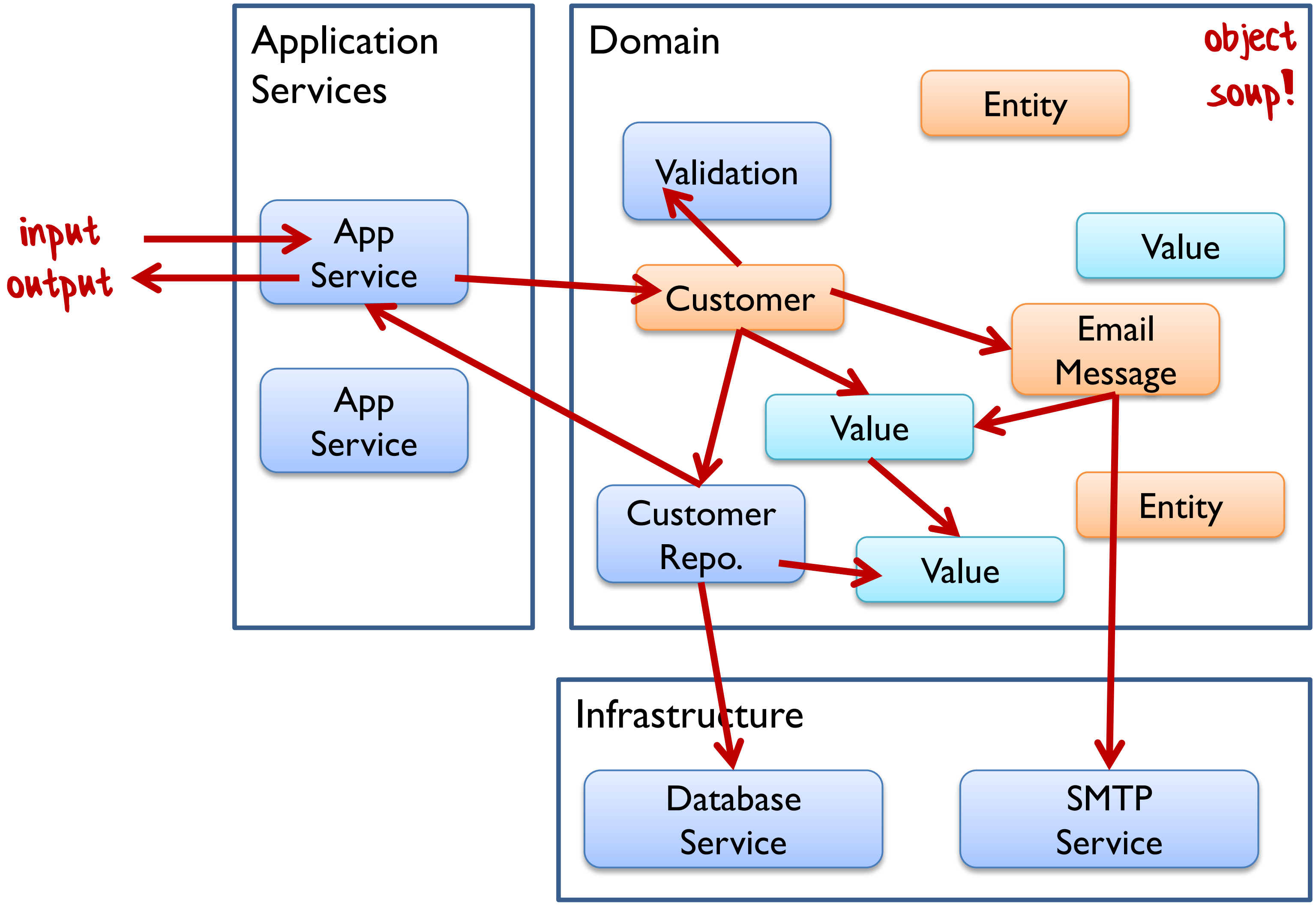
Flow of control in a FP use case



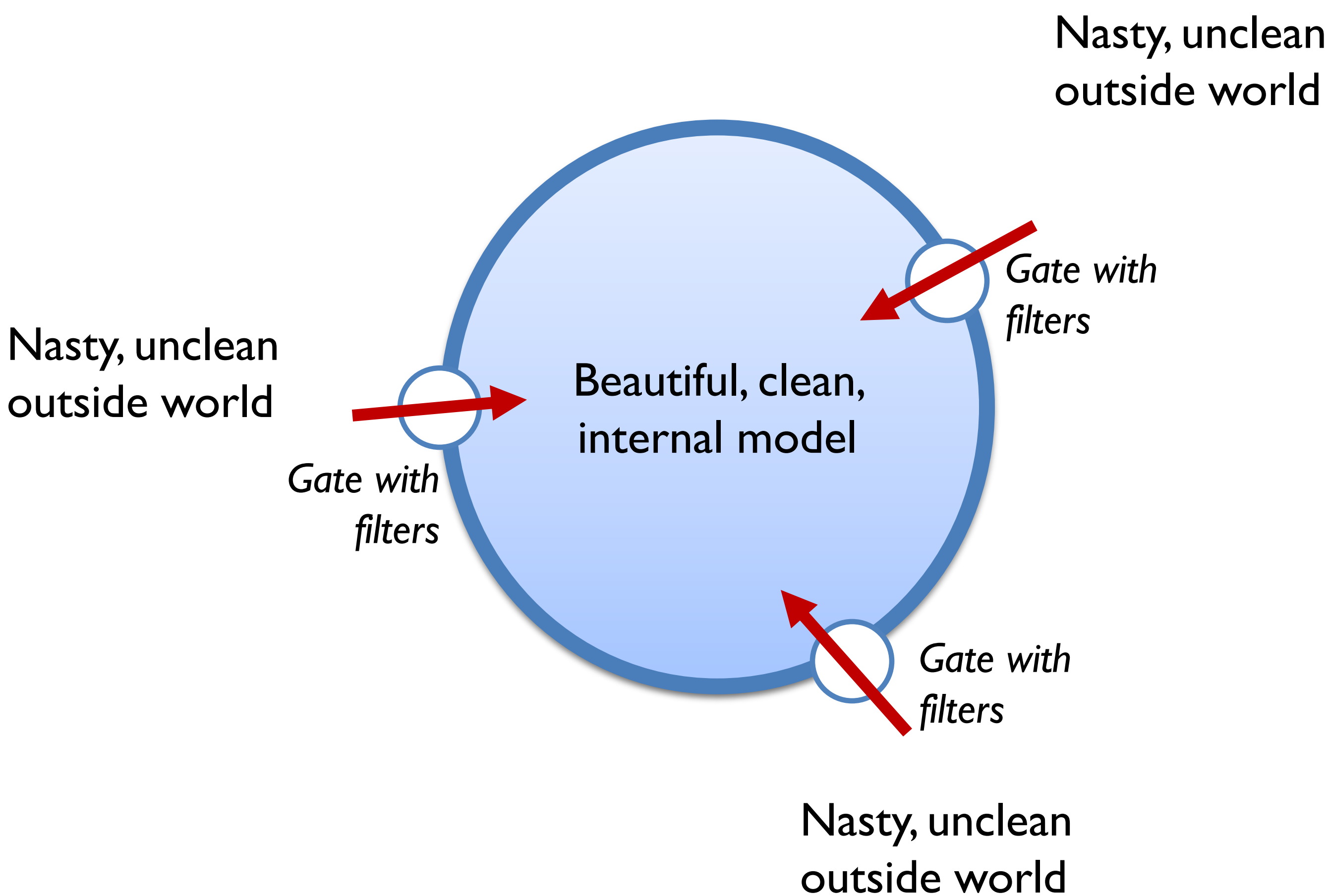
Linear!

Works well with domain events, FRP, etc

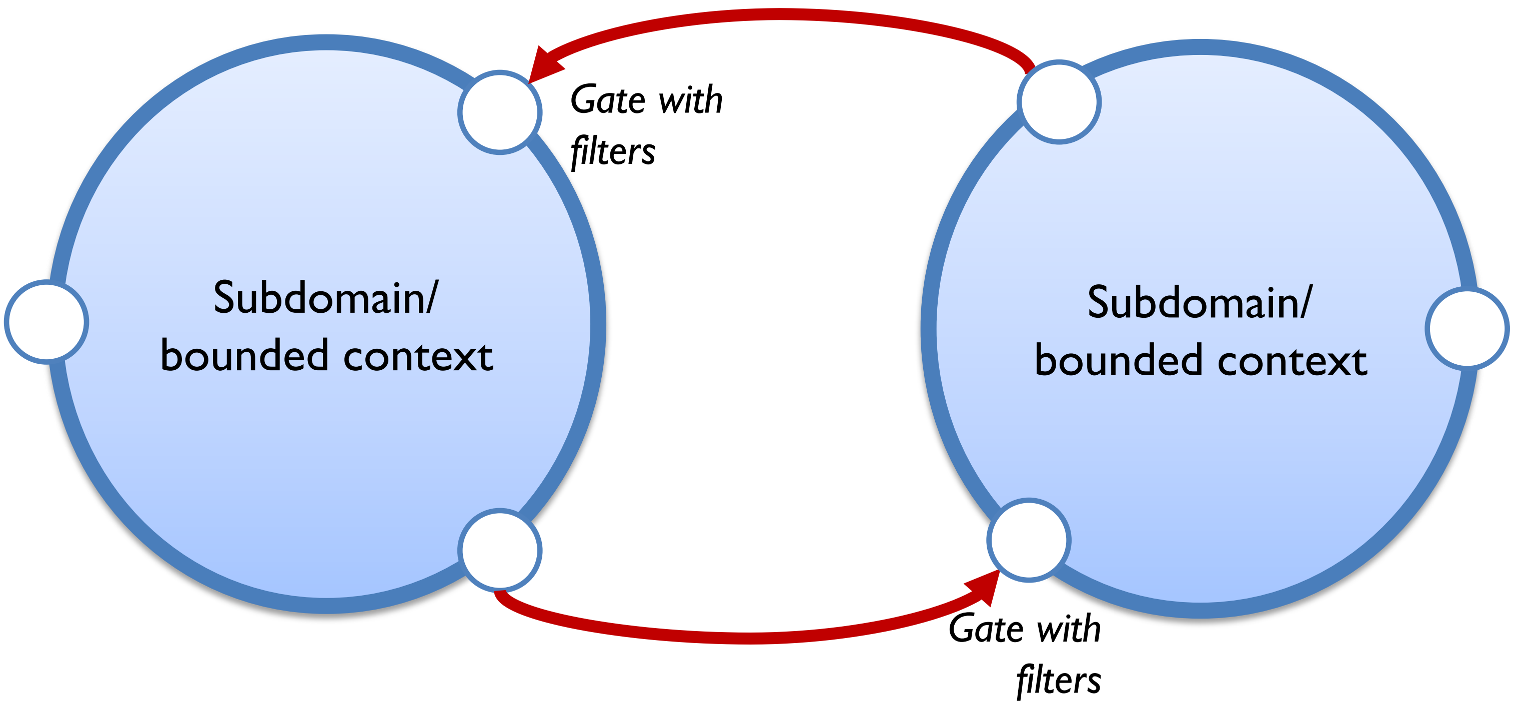
Flow of control in a OO use case



Interacting with the outside world

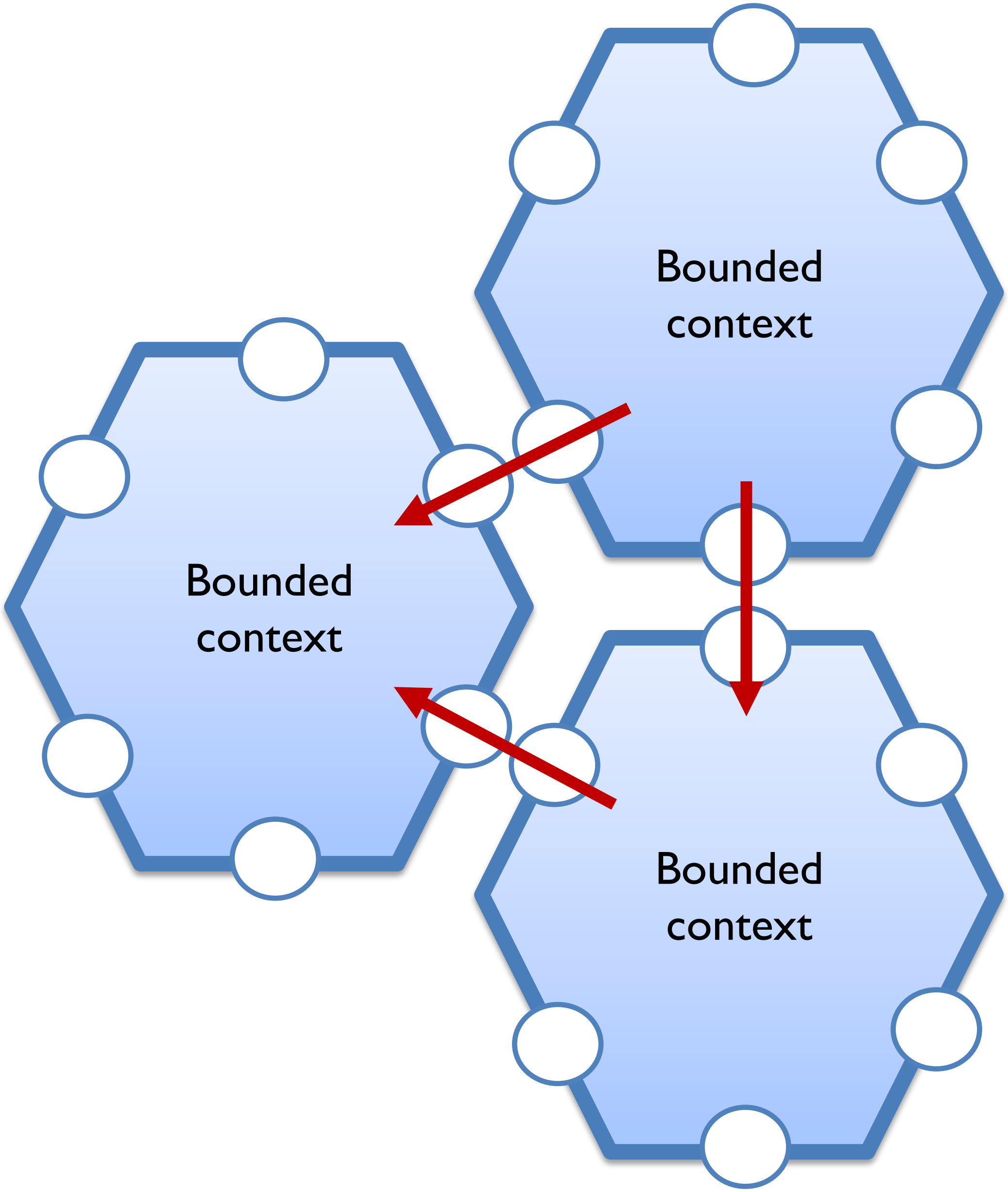


Interacting with the other domains



Transformation from one domain to another

Interacting with the other domains



Hexagonal architecture!

FUNCTIONS AS PARAMETERS

Guideline:
Parameterize all the things

Parameterize all the things

```
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
```

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

A good language makes 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

Initial Value

```
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
```

Parameterized action

Common code extracted

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

Guideline:

Be as generic as possible

Generic code

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

// val printList :
//   ('a -> unit) -> seq<'a> -> unit
```

Any kind of collection,
any kind of action!

F# and other functional languages
make code generic automatically

Generic code

```
int -> int
```

How many ways are there to implement this function?

```
'a -> 'a
```

How many ways are there to this function?

Generic code

```
int list -> int list
```

How many ways are there to implement this function?

```
'a list -> 'a list
```

How many ways are there to this function?

Generic code

```
('a -> 'b) -> 'a list -> 'b list
```

How many ways are there to implement this function?

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 * 2        // int -> int
```

Strategy pattern is trivial in FP

Object-oriented strategy pattern:

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

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

Functional equivalent:

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

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

Decorator pattern in FP

Functional equivalent of decorator pattern

```
let add1 x = x + 1    // int -> int
```

Decorator pattern in FP

Functional equivalent of decorator pattern

```
let add1 x = x + 1    // int -> int

let logged f x =
   printfn "input is %A" x
   let result = f x
   printfn "output is %A" result
   result
```

Decorator pattern in FP

Functional equivalent of decorator pattern

```
let add1 x = x + 1    // int -> int

let logged f x =
   printfn "input is %A" x
   let result = f x
   printfn "output is %A" result
   result

let add1Decorated =   // int -> int
   logged add1

[1..5] |> List.map add1
[1..5] |> List.map add1Decorated
```

Tip

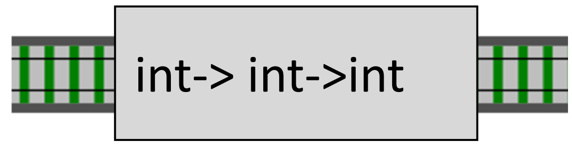
Every function is a

one parameter function

Writing functions in different ways

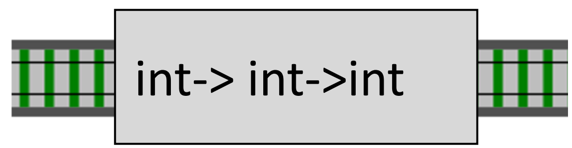
Two parameters

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



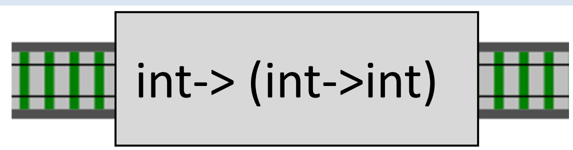
No parameters

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



One parameter

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



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

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

Missing a parameter?

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

```
let add1ToEach = List.map add1
```

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
```

```
let names = ["Alice"; "Bob"; "Scott"]
Names |> List.iter hello
```

One parameter

Pattern:
Use partial application to do dependency injection

Persistence agnostic

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

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

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

```
let getCustomer1 = getCustomerFromDatabase myConnection

// getCustomer1 : CustomerId -> Customer
```

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

```
let getCustomerFromMemory map (customerId:CustomerId) =
    map |> Map.find customerId
```

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

```
let getCustomer2 = getCustomerFromMemory inMemoryMap

// getCustomer2 : CustomerId -> Customer
```

Pattern:
The Hollywood principle: continuations

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

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!

Continuations

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

```
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
```

setup the functions to print a message

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

MONADS

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!

Let's do a cut & paste refactoring

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
```

Pyramid of doom: option example

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

let example input =
    let x = doSomething input
    if x.IsSome then
        doWithX x
    else
        None
```

Pyramid of doom: option example

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

let example input =
    let x = doSomething input
    if x.IsSome then
        doWithX x
    else
        None
```

Pyramid of doom: option example

```
let doWithY y =
    let z = doAThirdThing y
    if z.IsSome then
        let result = z.Value
        Some result
    else
        None

let doWithX x =
    let y = doSomethingElse x
    if y.IsSome then
        doWithY y
    else
        None

let example input =
    let x = doSomething input
    if x.IsSome then
        doWithX x
    else
        None
```

Pyramid of doom: option example refactored

```
let doWithZ z =
    let result = z
    Some result

let doWithY y =
    let z = doAThirdThing y
    if z.IsSome then
        doWithZ z.Value
    else
        None

let doWithX x =
    let y = doSomethingElse x
    if y.IsSome then
        doWithY y.Value
    else
        None

let optionExample input =
    let x = doSomething input
    if x.IsSome then
        doWithX x.Value
    else
        None
```

Three small pyramids instead of one big one!

This is still ugly!

But the code has a pattern...

Pyramid of doom: option example refactored

```
let doWithZ z =
    let result = z
    Some result

let doWithY y =
    let z = doAThirdThing y
    if z.IsSome then
        doWithZ z.Value
    else
        None

let doWithX x =
    let y = doSomethingElse x
    if y.IsSome then
        doWithY y.Value
    else
        None

let optionExample input =
    let x = doSomething input
    if x.IsSome then
        doWithX x.Value
    else
        None
```

But the code has a pattern...

```
let doWithY y =
    let z = doAThirdThing y
    if z.IsSome then
        doWithZ z.Value
    else
        None
```

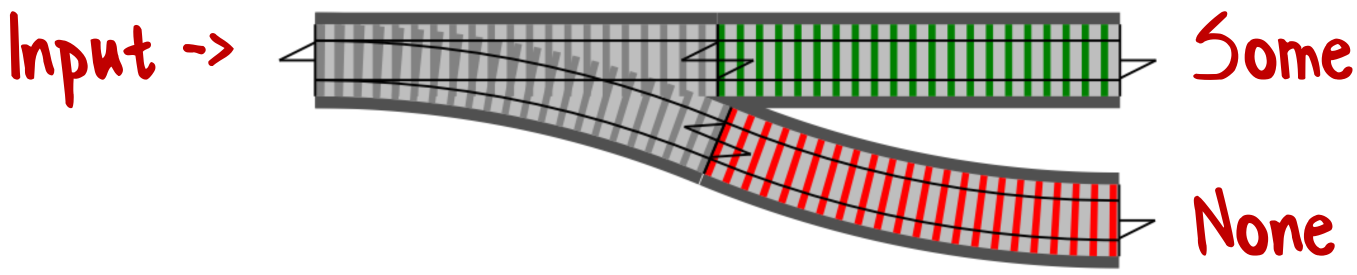
Crying out to be parameterized!

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


let doWithY y =
    let z = doAThirdThing y
    z |> ifSomeDo doWithZ
```

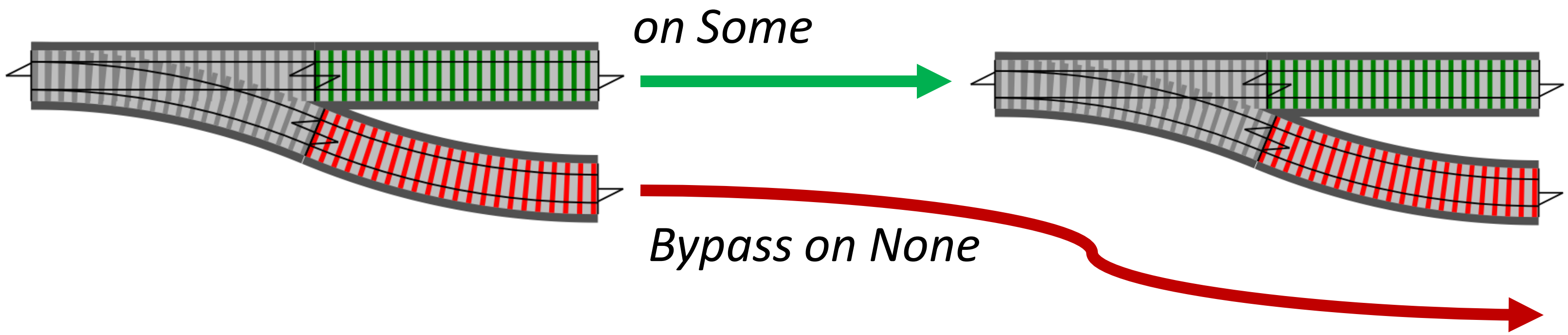
```
let ifSomeDo f x =
    if x.IsSome then
        f x.Value
    else
        None


let doWithY y =
    y
    |> doAThirdThing
    |> ifSomeDo doWithZ
```

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


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

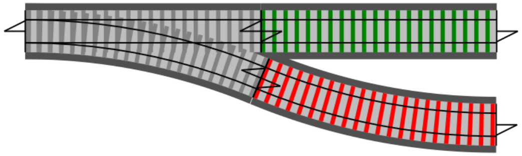
A switch analogy

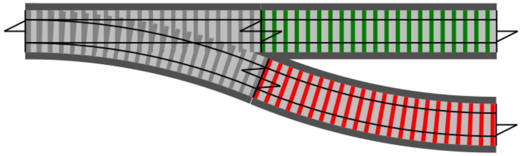


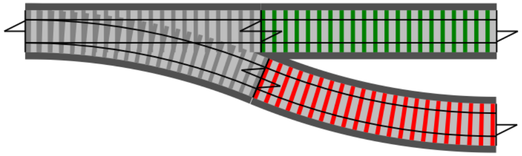
Connecting switches



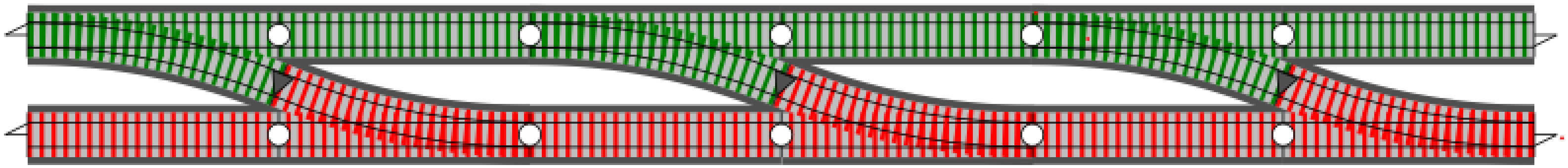
Connecting switches



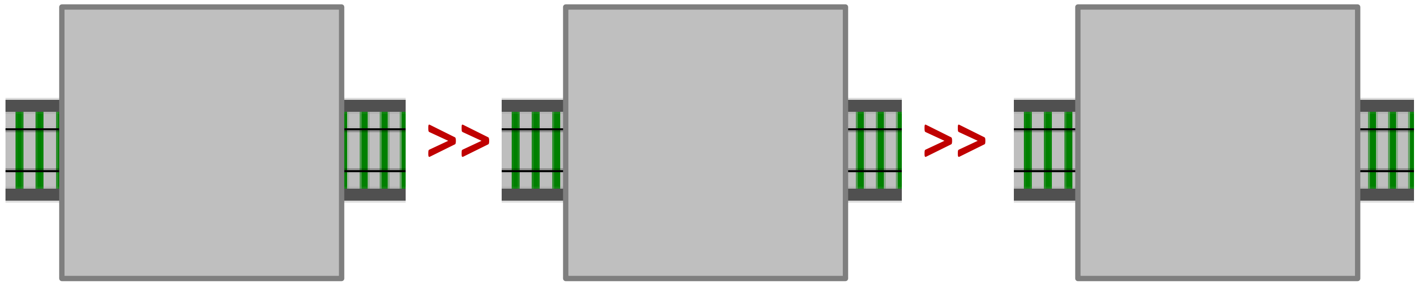




Connecting switches

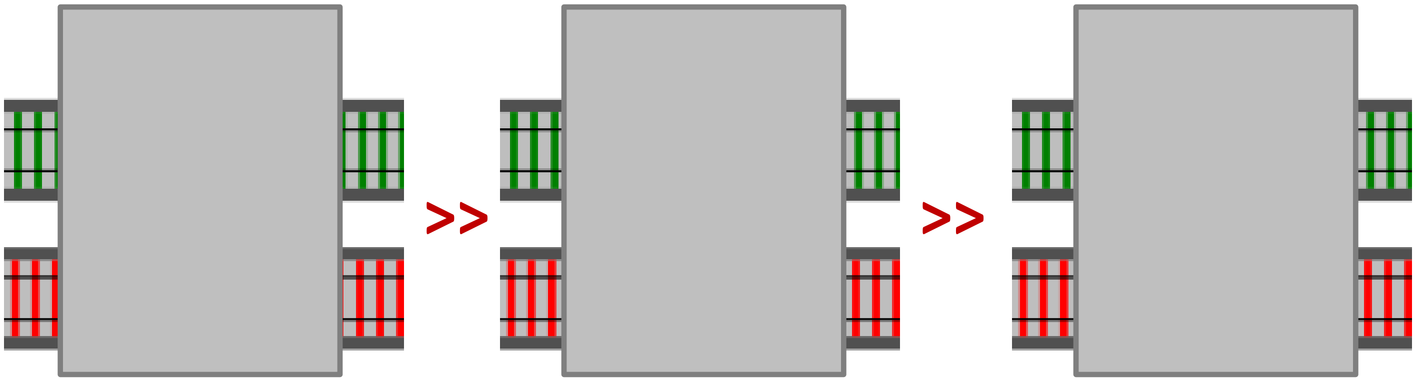


Composing switches



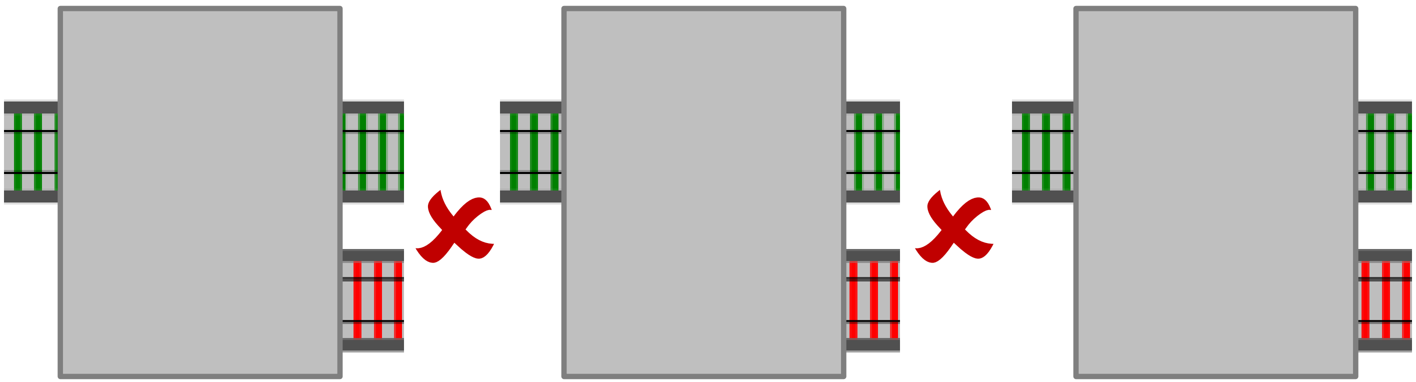
Composing one-track functions is fine...

Composing switches



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

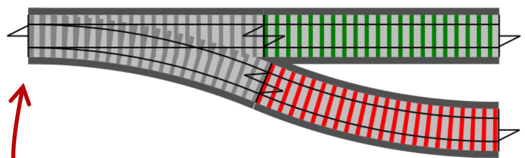
Composing switches



... but composing switches is not allowed!

Composing switches

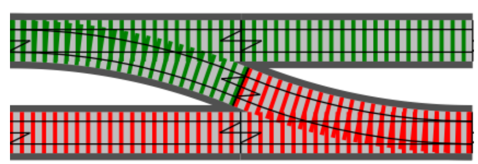
One-track input



Two-track input

Before: Not suitable for composition

Two-track input

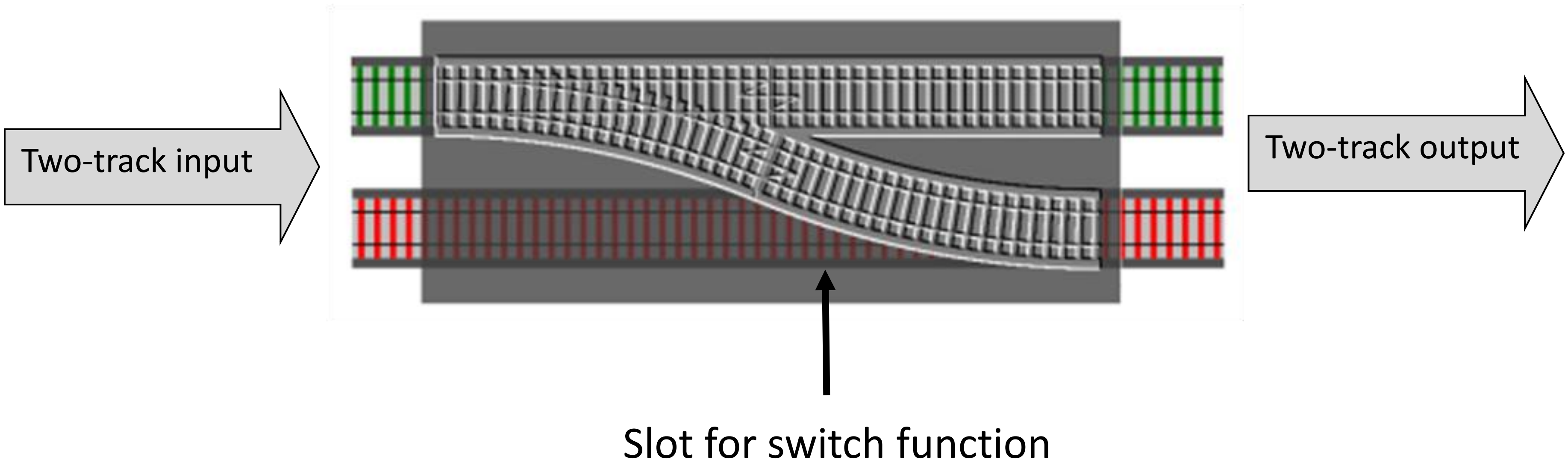


Two-track input

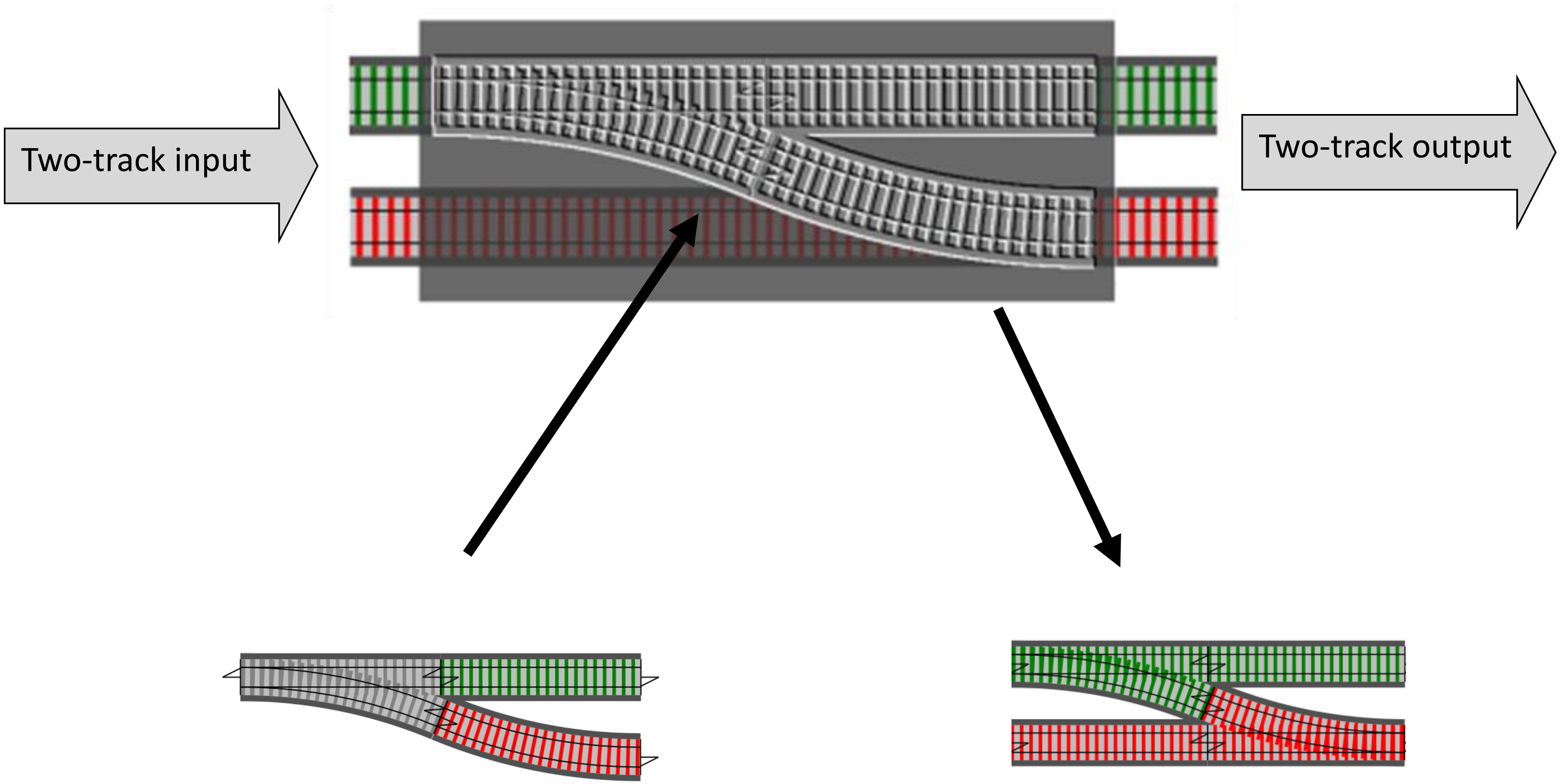
After: Suitable for composition

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

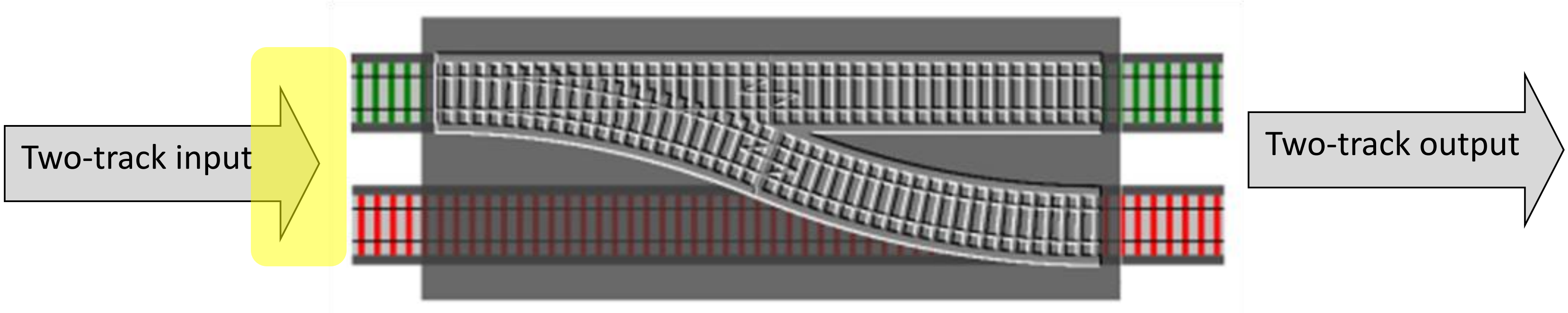
Building an adapter block



Building an adapter block

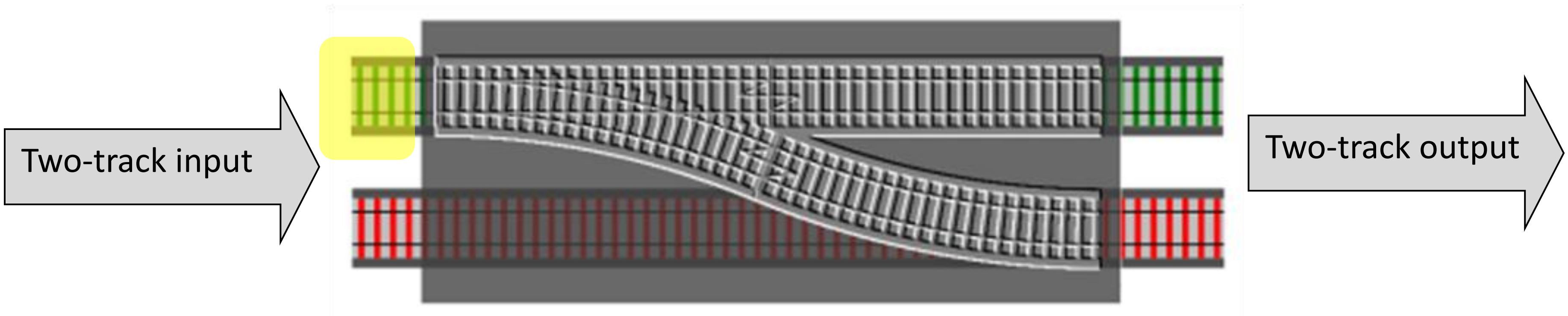


Building an adapter block



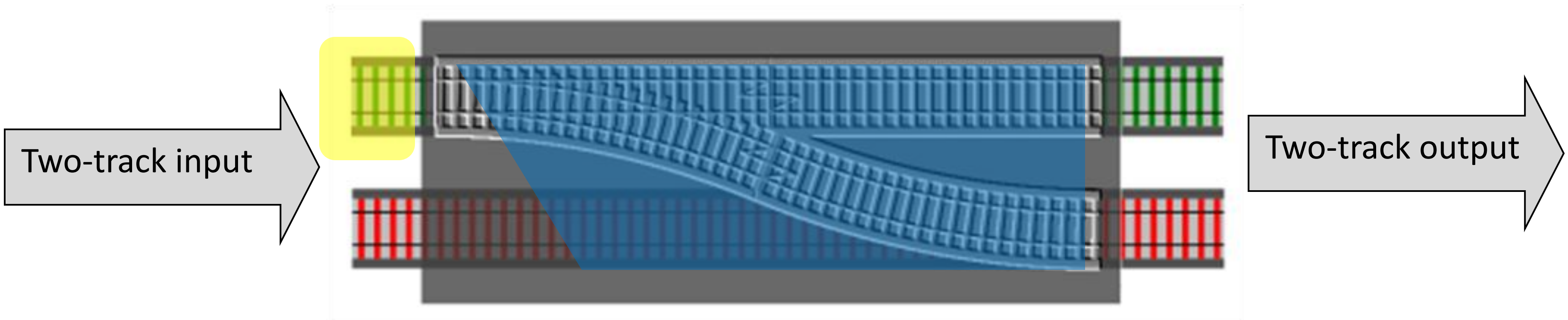

```
let bind nextFunction optionInput =
   match optionInput with
   | Some s -> nextFunction s
   | None -> None
```

Building an adapter block




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

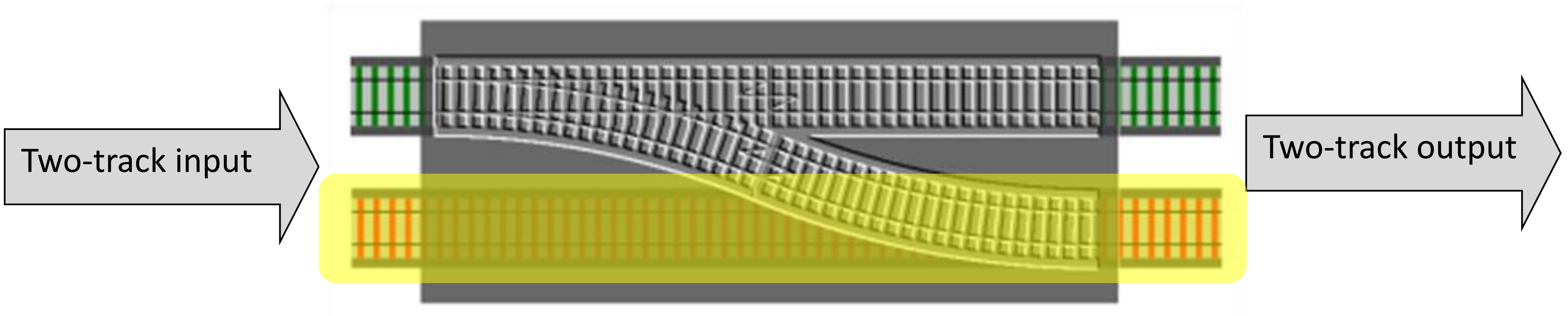
Building an adapter block




```
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)
```

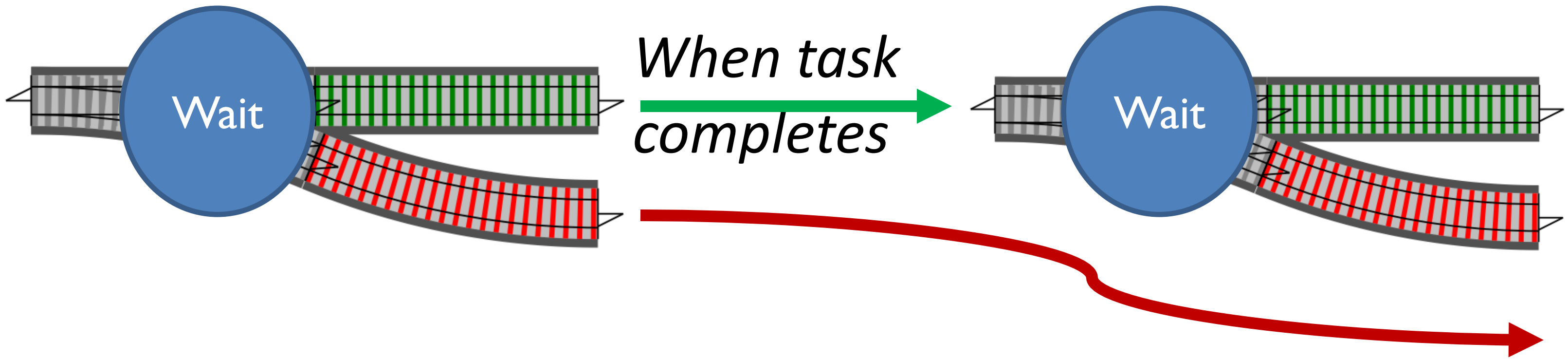
No pyramids!

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

This pattern is also a "monadic bind"

Computation expressions

```
let example input =
    maybe {
    let! x = doSomething input
    let! y = doSomethingElse x
    let! z = doAThirdThing y
    return z
    }
```

Computation expression

```
let taskExample input =
    task {
    let! x = startTask input
    let! y = startAnotherTask x
    let! z = startThirdTask z
    return z
    }
```

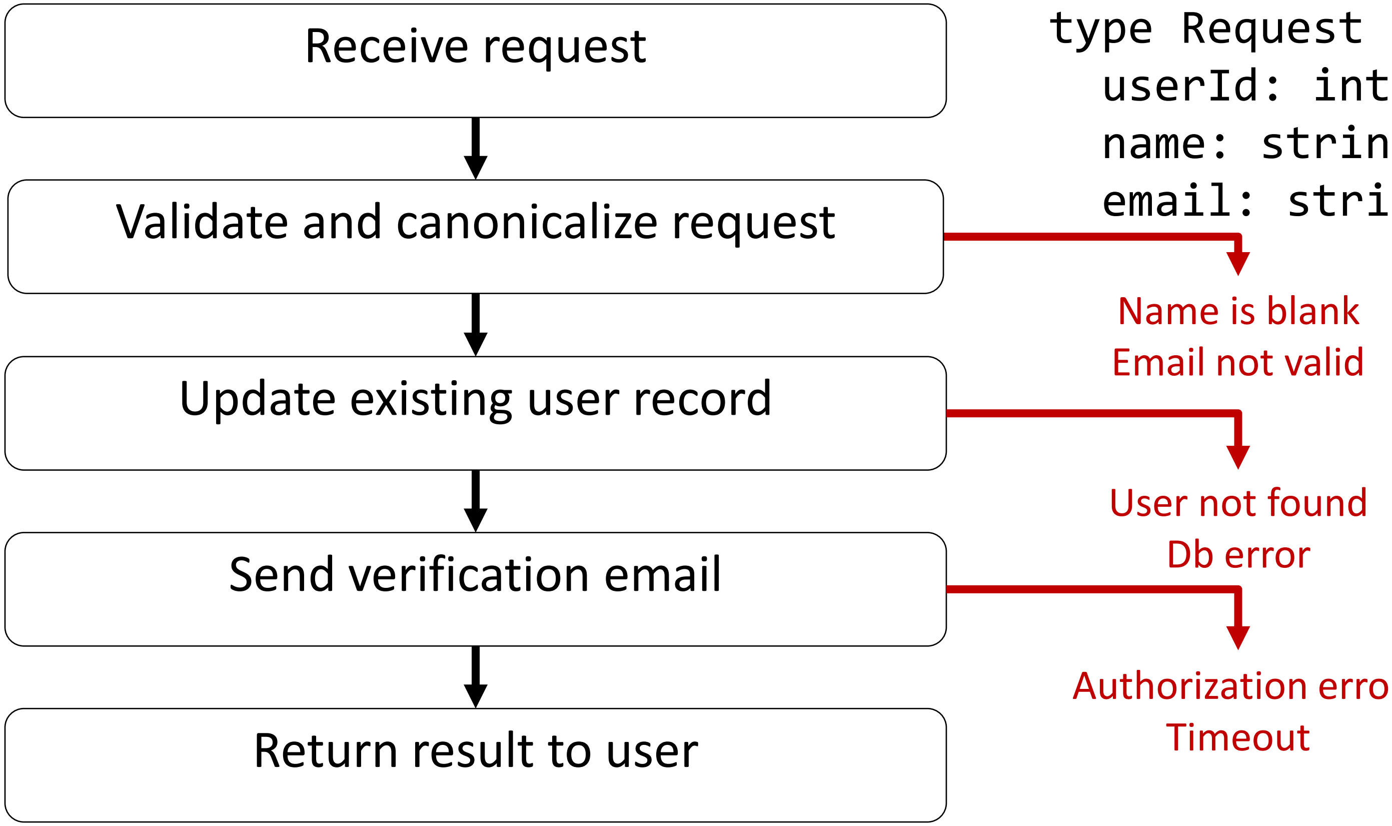
Computation expression

Pattern:

Use bind to chain error handlers

Example use case

"As a user I want to update my name and email address"
- and see sensible error messages when something goes wrong!




```
type Request = {
  userId: int;
  name: string;
  email: string }
```

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";
}
```

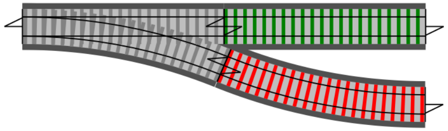
6 clean lines -> 18 ugly lines. 200% extra!
Sadly this is typical of error handling code.

A structure for managing errors

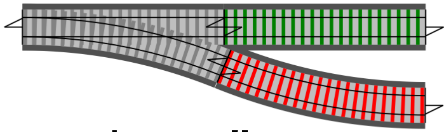
```
type TwoTrack<'TEntity> =
   | Success of 'TEntity
   | Failure of string
```


```
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
```

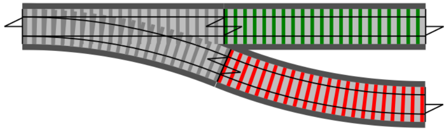
Bind example




```
let nameNotBlank input =
  if input.name = "" then
    Failure "Name must not be blank"
  else Success input
```

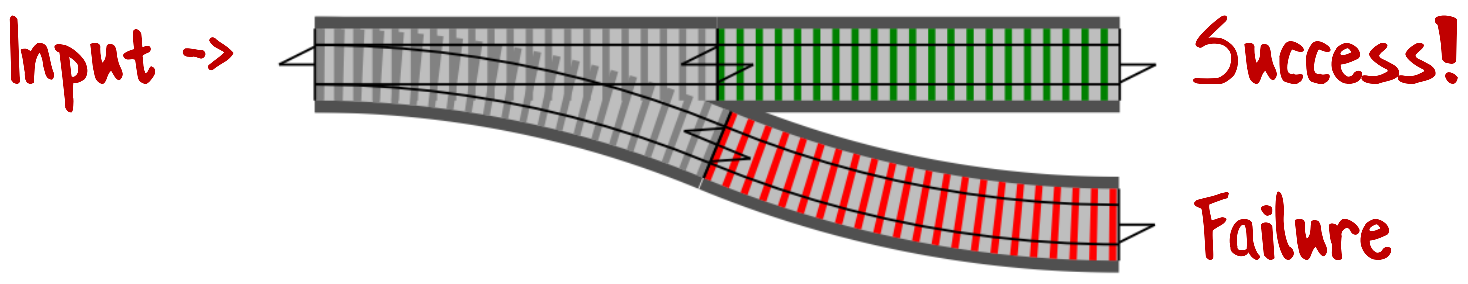



```
let name50 input =
  if input.name.Length > 50 then
    Failure "Name must not be longer than 50 chars"
  else Success input
```

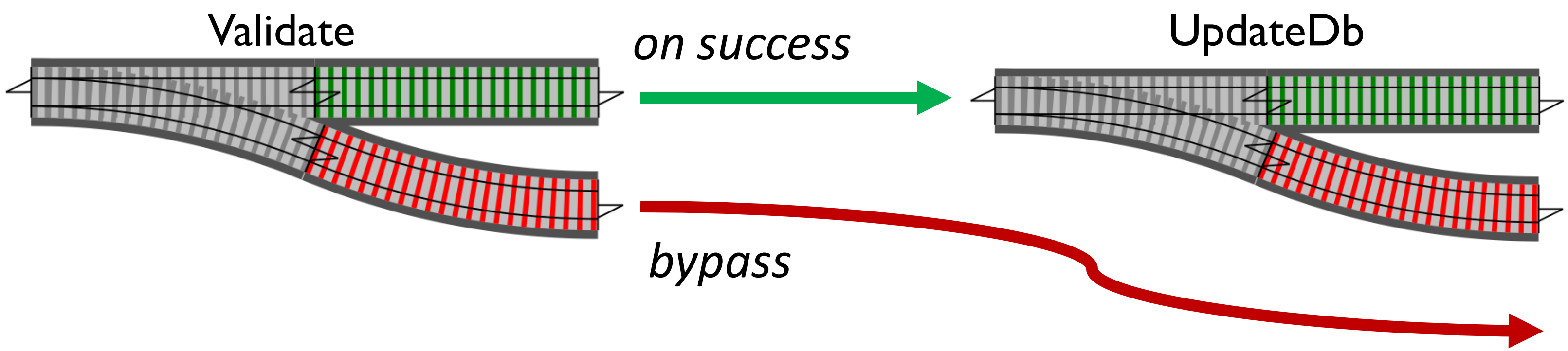



```
let emailNotBlank input =
  if input.email = "" then
    Failure "Email must not be blank"
  else Success input
```

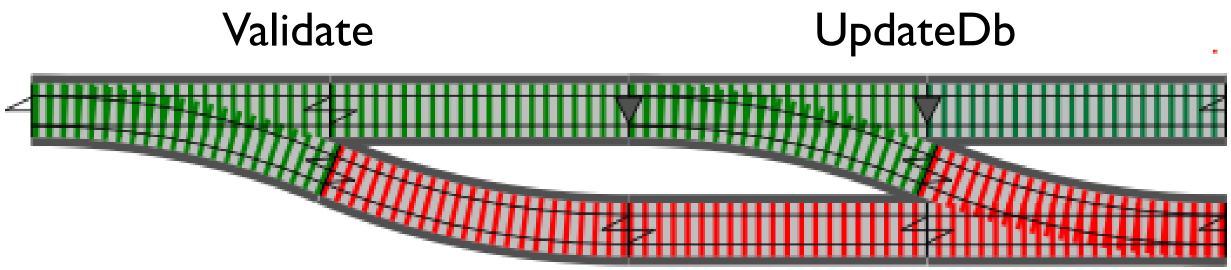
Switches again



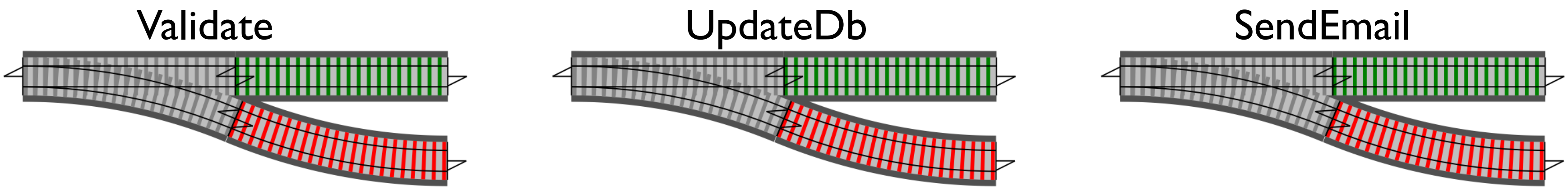
Connecting switches



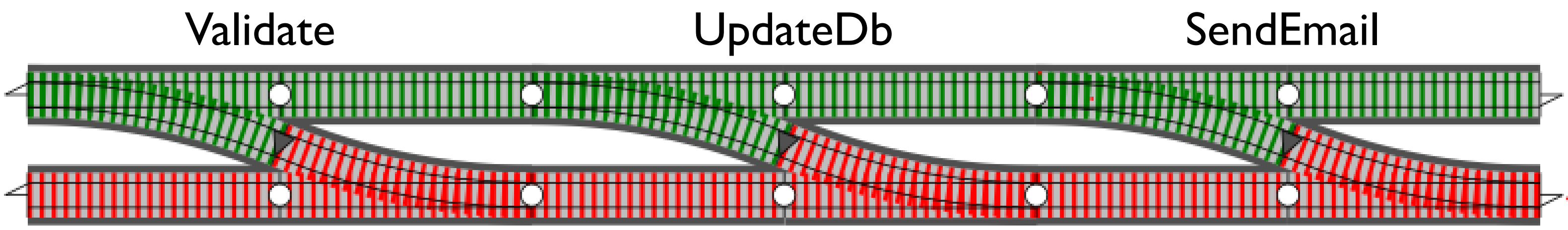
Connecting switches



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
```

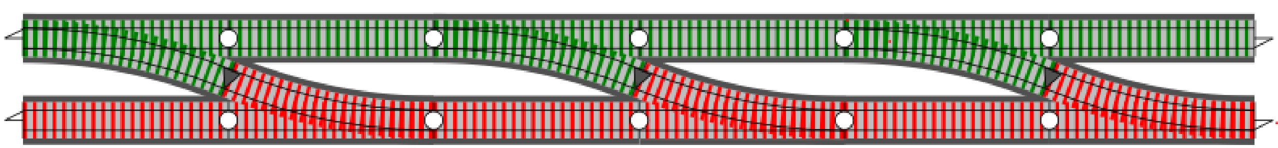
One track

Functional flow with error handling

After

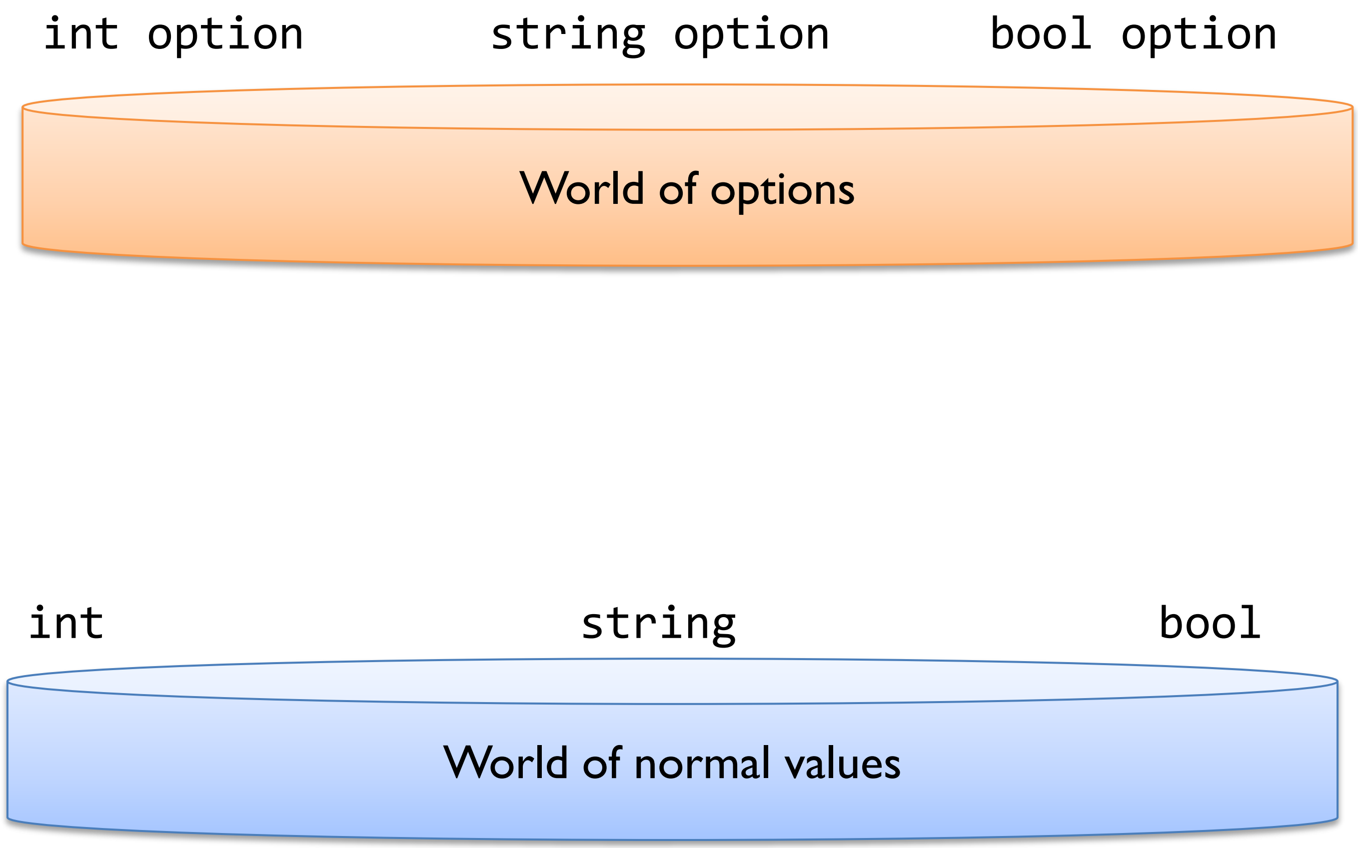
```
let updateCustomerWithErrorHandling =
   receiveRequest
   |> validateRequest
   |> canonicalizeEmail
   |> updateDbFromRequest
   |> sendEmail
   |> returnMessage
```

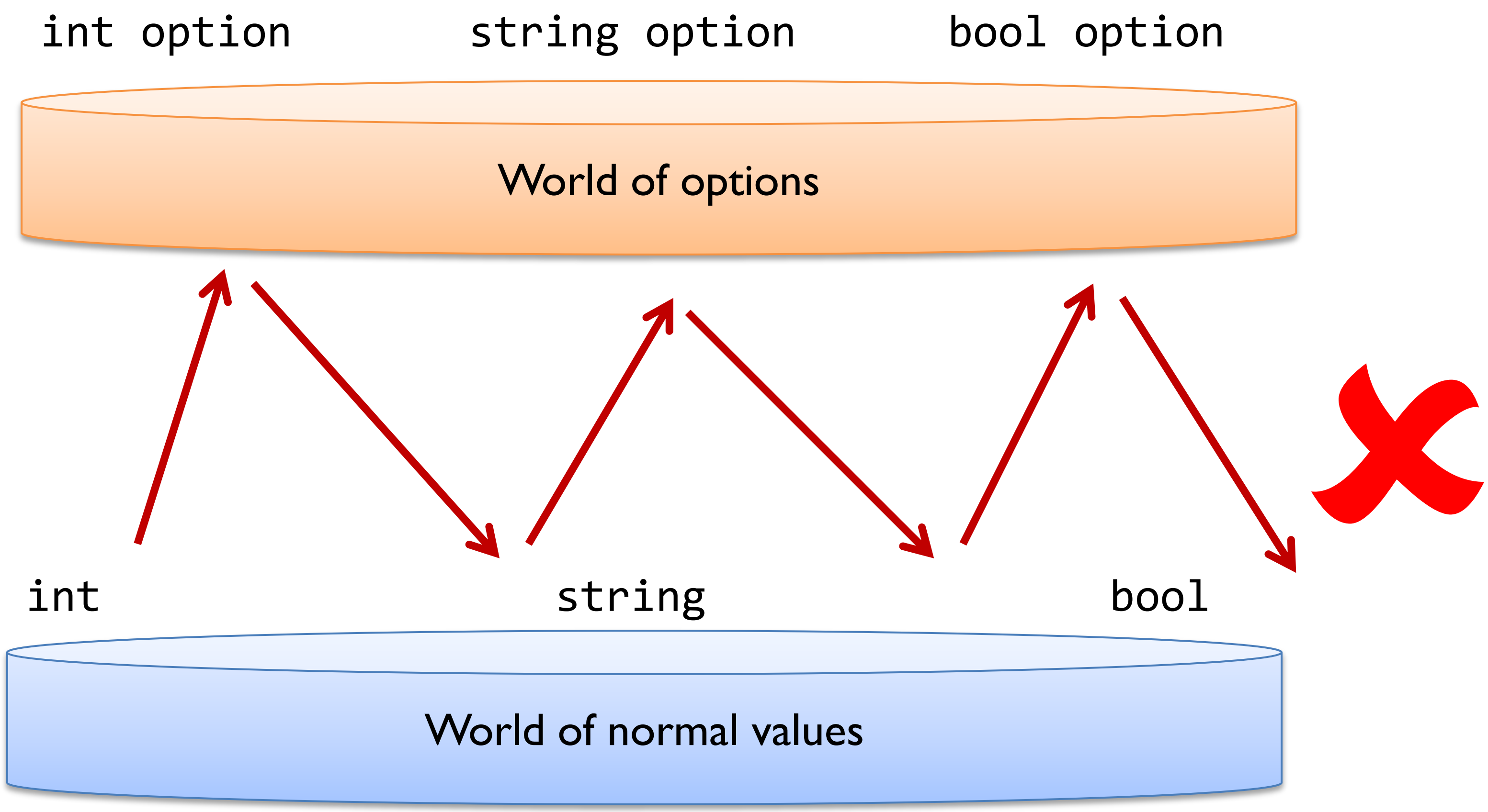
Two track

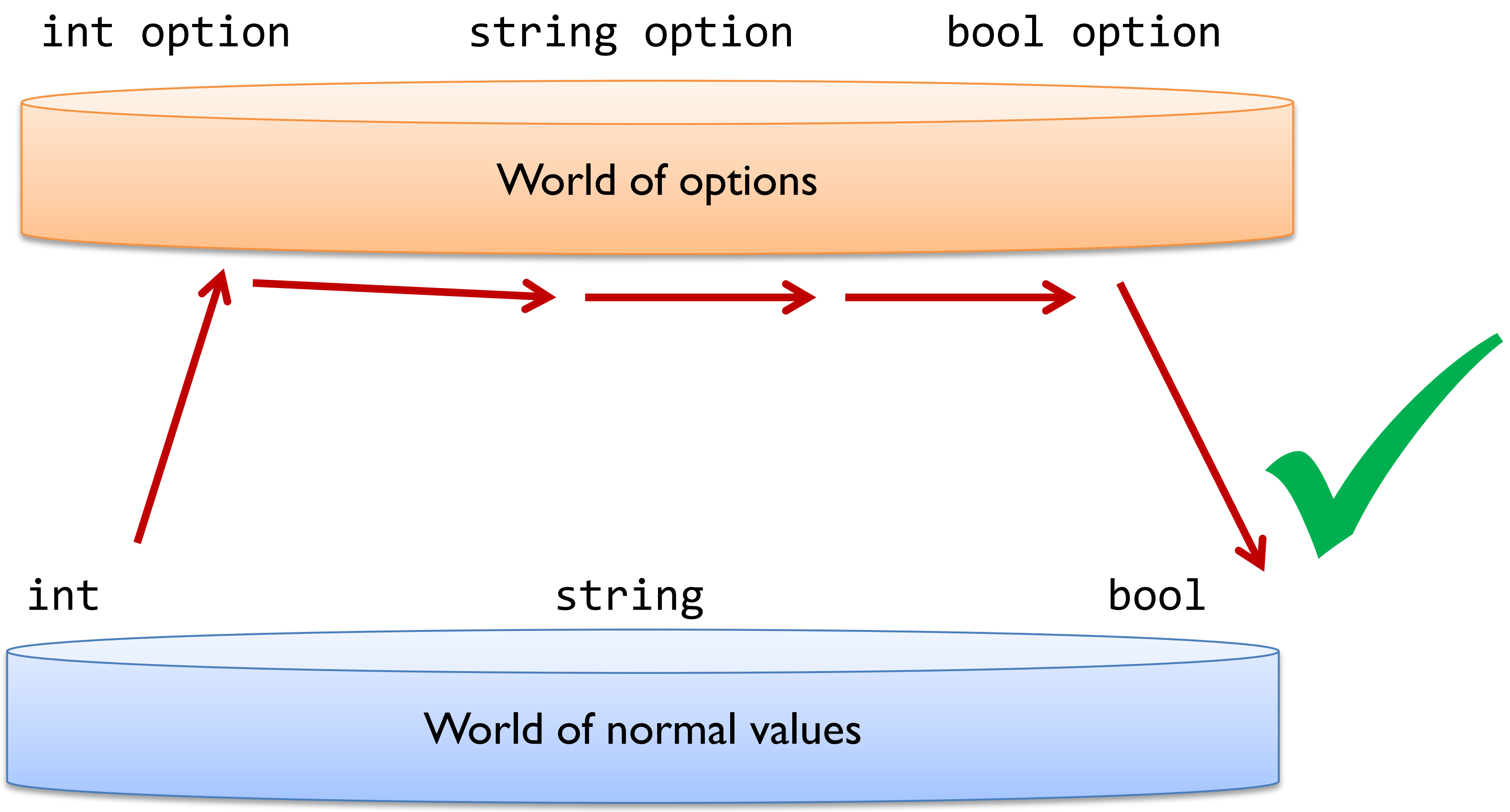


See fsharpforfunandprofit.com/rop

MAPS AND APPLICATIVES


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


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


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

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:

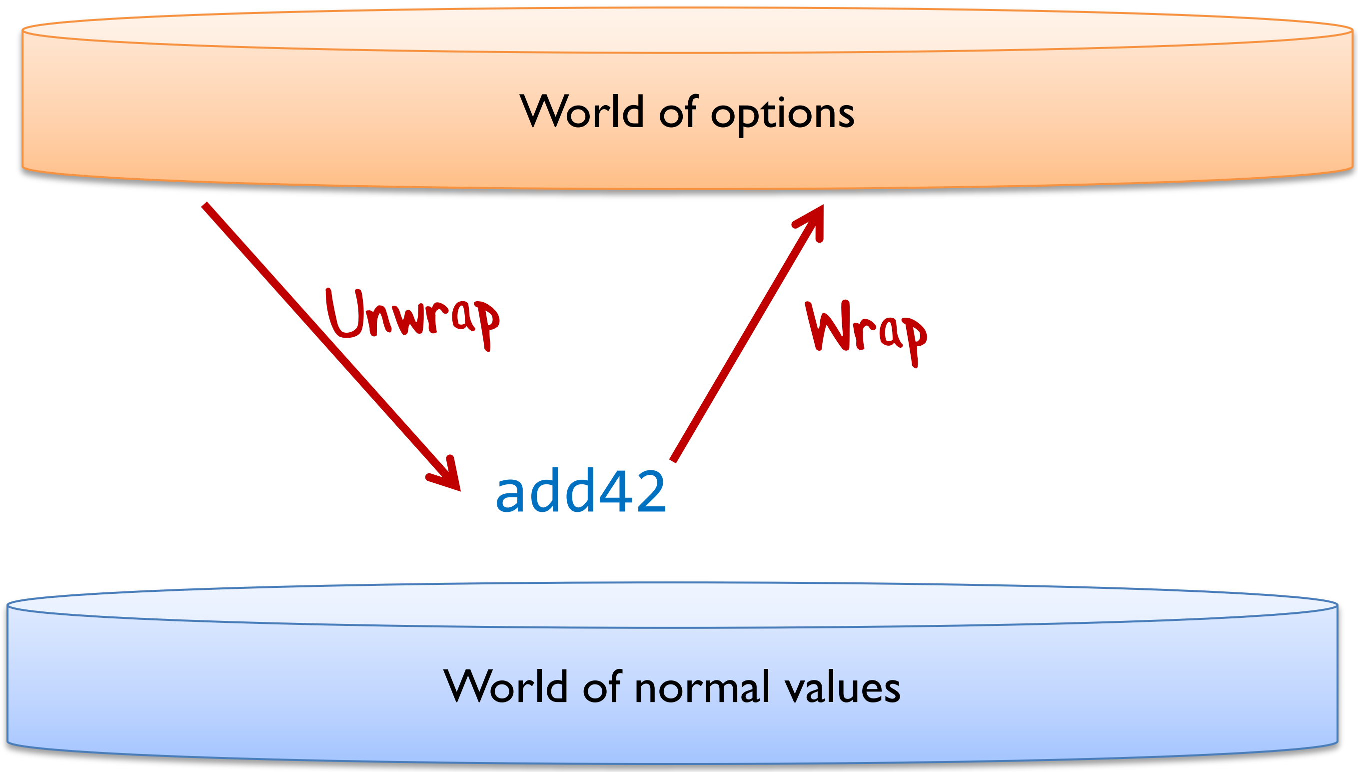
Works on normal values

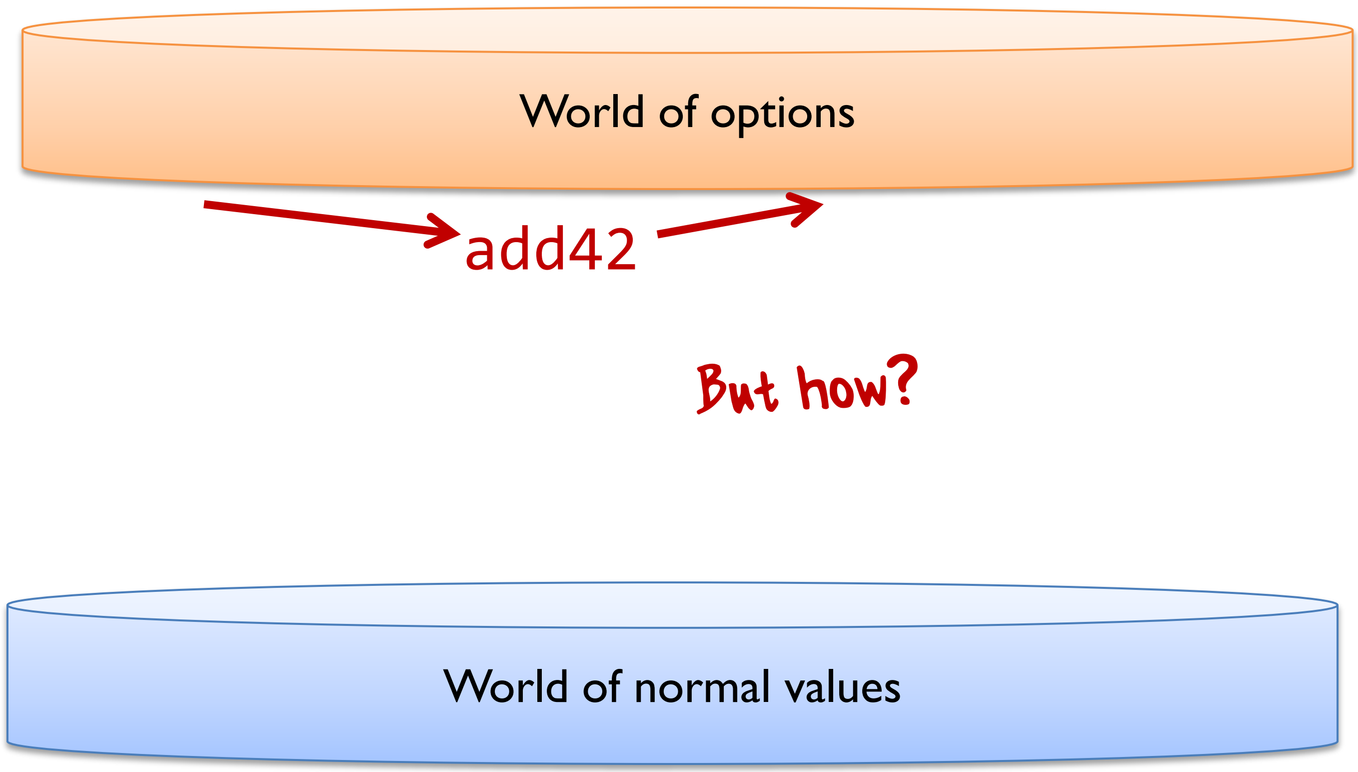
```
let add42 x = x + 42
```

Unwrap

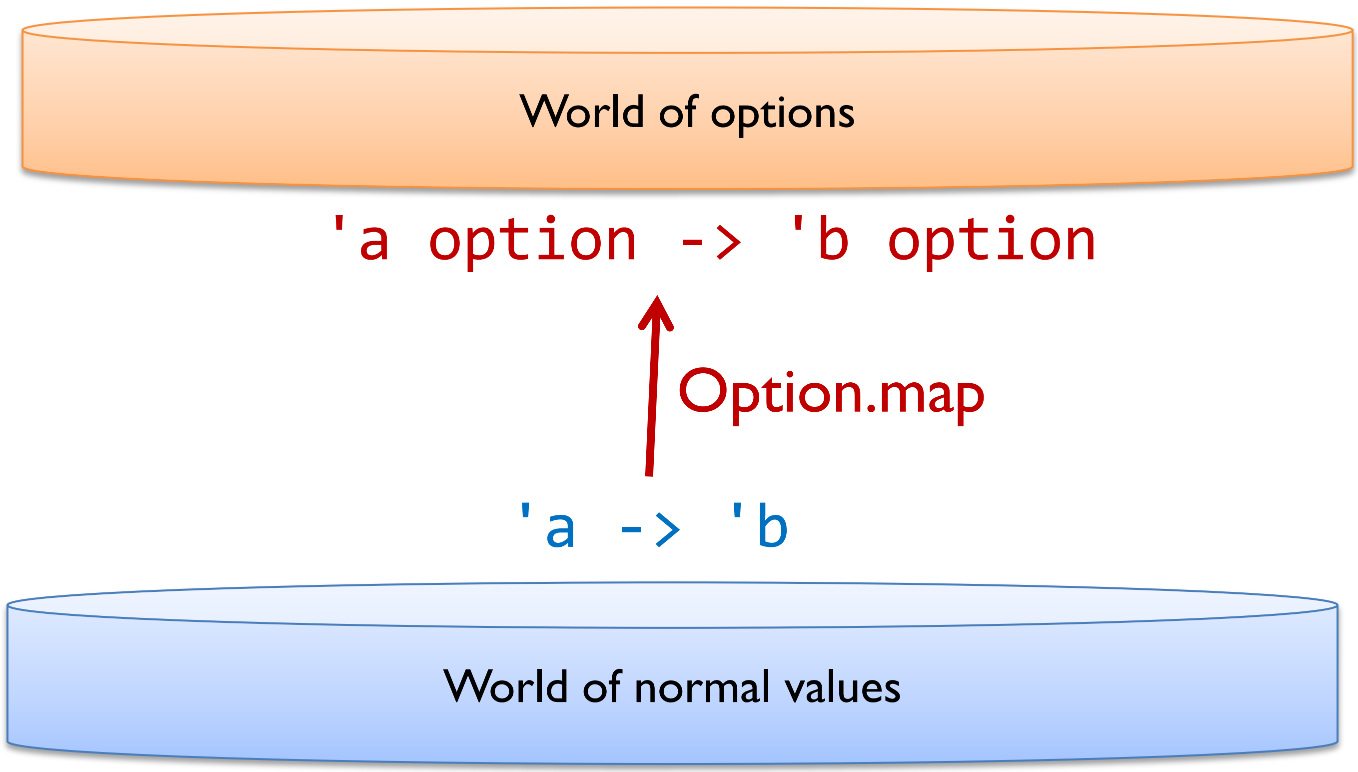
```
let add42ToOption opt =
    if opt.IsSome then
        let newVal = add42 opt.Value
        Some newVal
    else
        None
```

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
'a option -> 'b option
Option.map
'a -> 'b
World of normal values

The right way to code with options

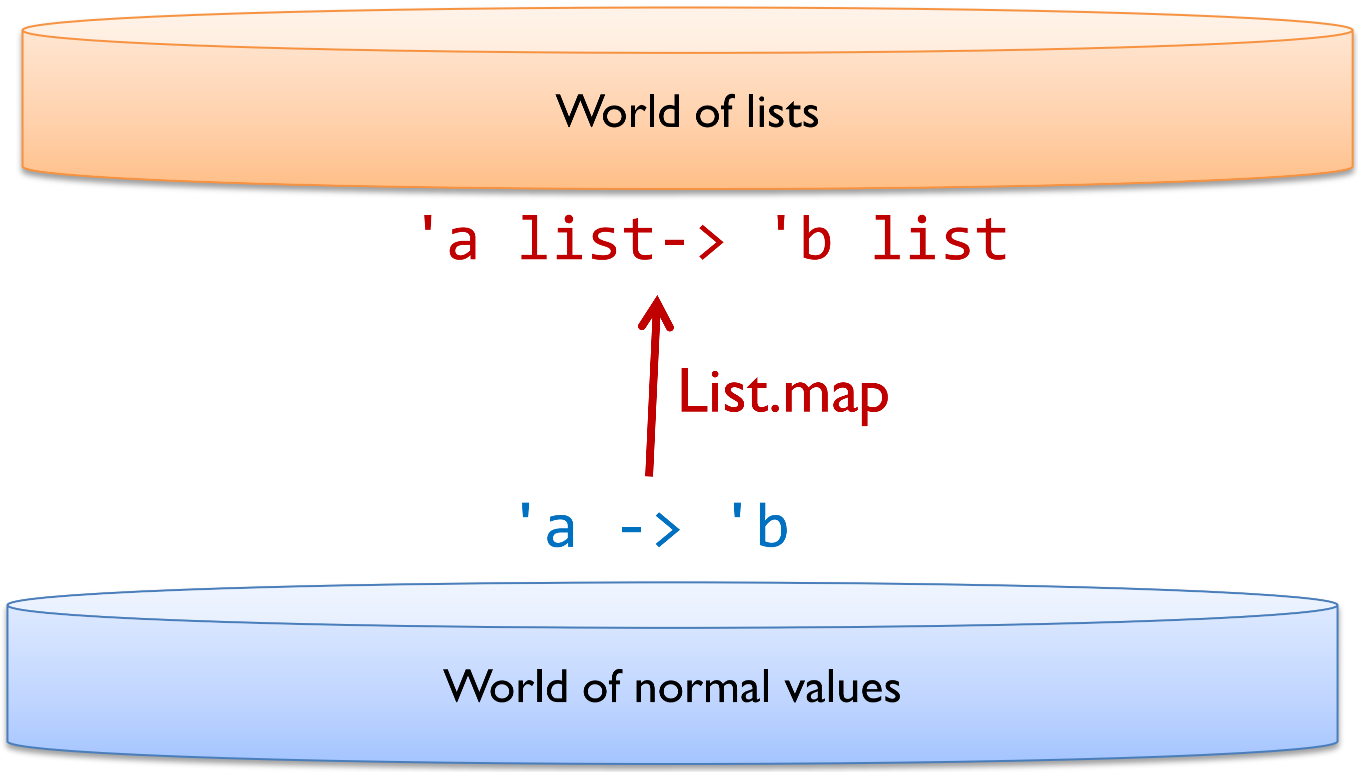
Let's say you have an int wrapped in an Option, and you want to add 42 to it:

```
let add42 x = x + 42

let add42ToOption = Option.map add42
Some 1 |> add42ToOption

Some 1 |> Option.map add42
```

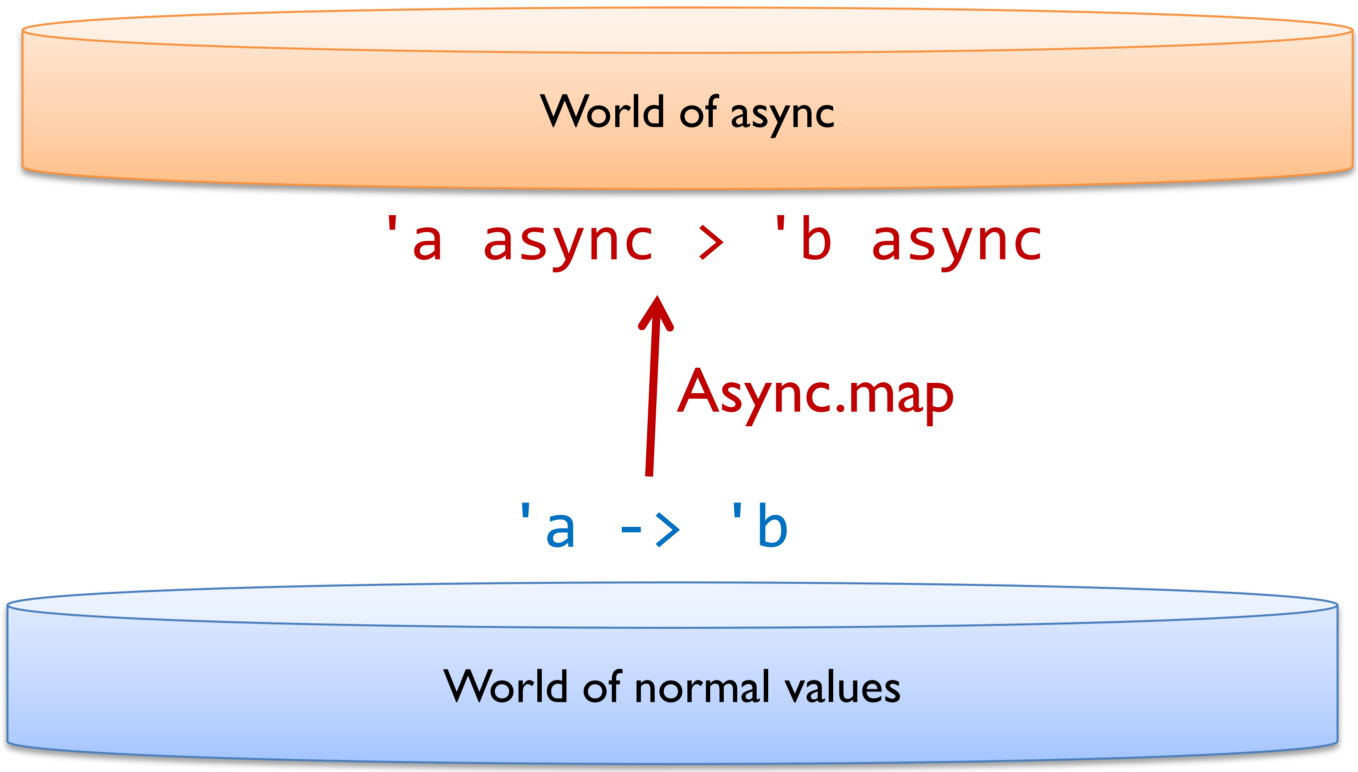
Often no need to bother creating a temp function

Pattern:
Use “map” to lift functions

Lifting to lists



Lifting to async



The right way to code with wrapped types

Most wrapped types provide a "map"

```
let add42 x = x + 42

Some 1 |> Option.map add42

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

Guideline:
If you create a wrapped generic type, create a "map" for it.

Terminology alert:
Mappable types are "functors"

Maps

```
type TwoTrack<'TEntity> =
   | Success of 'TEntity
   | Failure of string
```

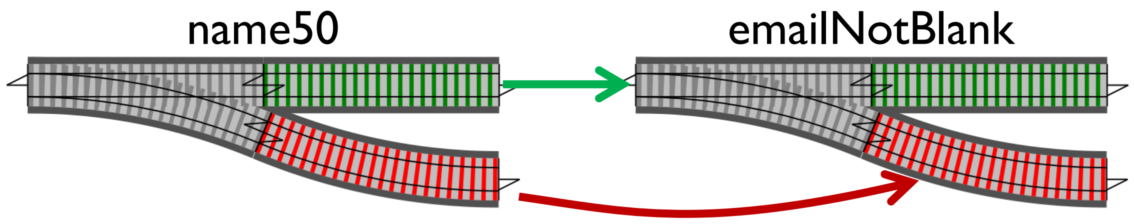
My Map!

```
let mapTT f x =
   match x with
   | Success entity -> Success (f entity)
   | Failure s -> Failure s
```

Tip:
Use applicatives for validation

What's an applicative?

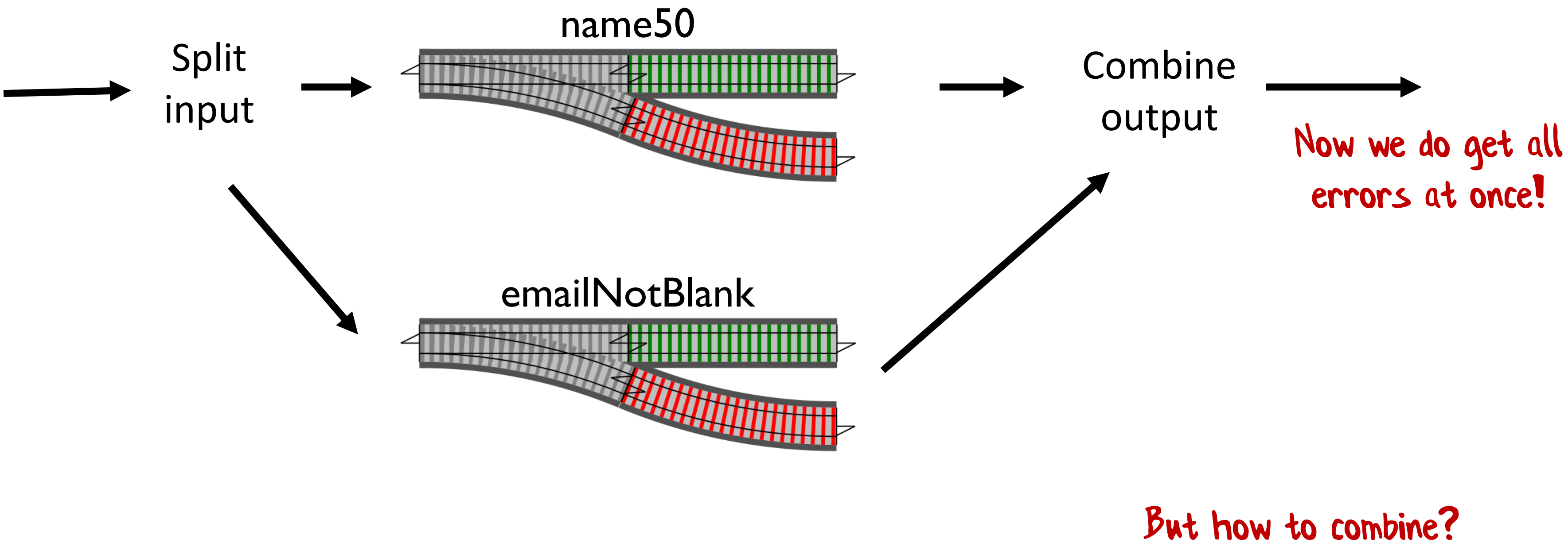
Series validation



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.

Parallel validation



Creating a valid data structure

Domain object

```
type Request= {
    UserId: UserId;
    Name: String50;
    Email: EmailAddress}
```

These are "pure"

Object from the outside

```
type RequestDTO= {
    UserId: int;
    Name: string;
    Email: string}
```

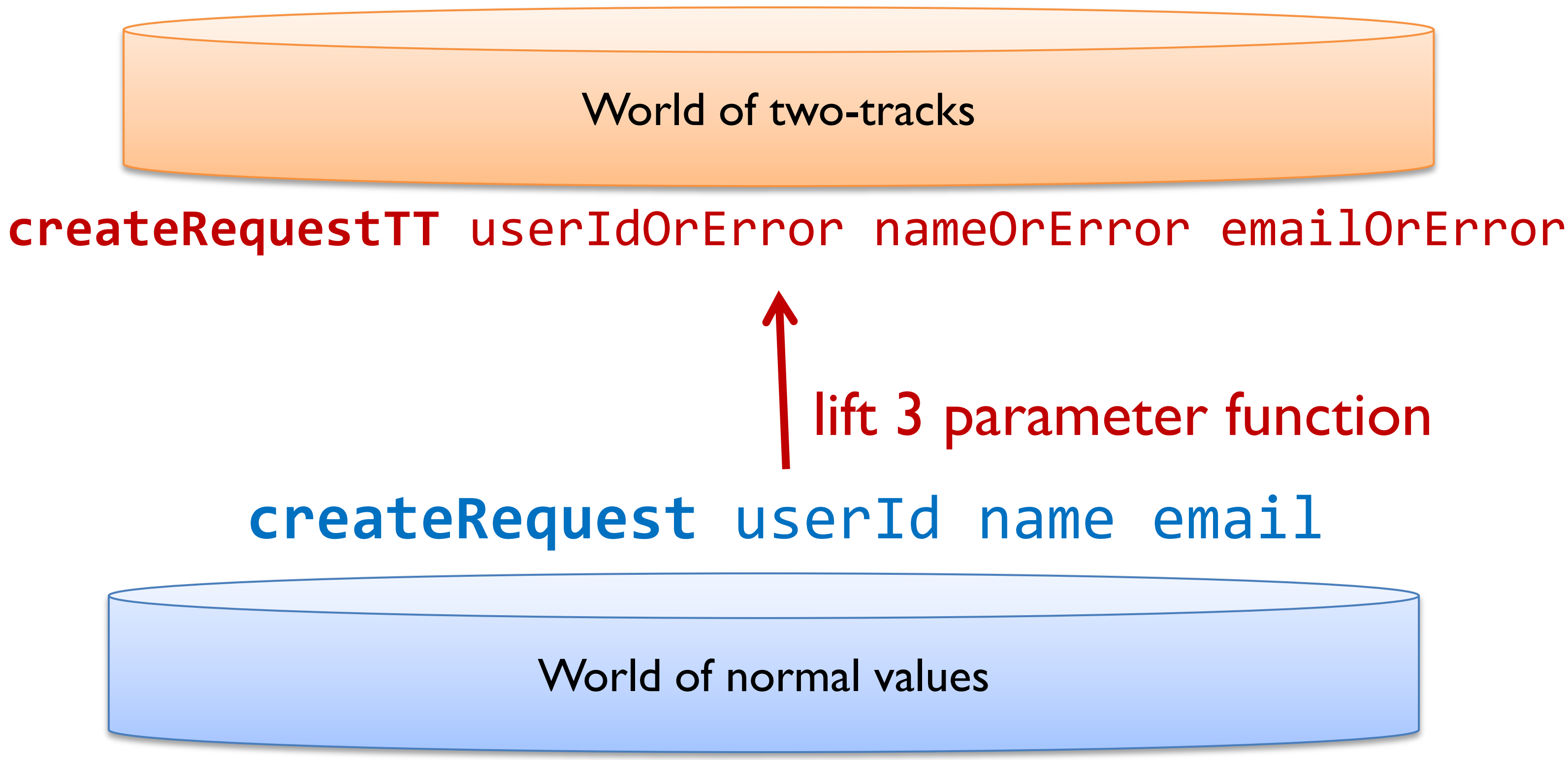
These need validation

How not to do validation

```
// do the validation of the DTO
let userIdOrError = validateUserId dto.userId
let nameOrError = validateName dto.name
let emailOrError = validateEmail dto.email

if userIdOrError.IsSuccess
   && nameOrError.IsSuccess
   && emailOrError.IsSuccess then
      {
      userId = userIdOrError.SuccessValue
      name = nameOrError.SuccessValue
      email = emailOrError.SuccessValue
      }
else if userIdOrError.IsFailure
   && nameOrError.IsSuccess
   && emailOrError.IsSuccess then
       userIdOrError.Errors
else ...
```

No no no this is horrible!

Lifting to TwoTracks



The right way

First create a function that creates a Request

```
let createRequest userId name email =
    {
    userId = userIdOrError.SuccessValue
    name = nameOrError.SuccessValue
    email = emailOrError.SuccessValue
    }
```

then "lift" it to the world of two track errors

```
let createRequestTT = lift3 createRequest
```

There is a separate "lift" function for each number of parameters: "lift2" "lift3" etc.

"lift1" is just "map"

The right way

```
let createRequestTT = lift3 createRequest
```

Now you can pass in the errors safely

```
let userIdOrError = validateUserId dto.userId
let nameOrError = validateName dto.name
let emailOrError = validateEmail dto.email

let requestOrError =
  createRequestTT userIdOrError nameOrError emailOrError
```

The result is also in the world of two tracks

The right way

Alternative using <!> and <*>

```
let userIdOrError = validateUserId dto.userId
let nameOrError = validateName dto.name
let emailOrError = validateEmail dto.email

let requestOrError =
  createRequest
      <!> userIdOrError
      <*> nameOrError
      <*> emailOrError
```

Guideline:
If you use a wrapped generic type, look for a set of "lifts" associated with it

Terminology alert:
Liftable types are "applicative functors"

Guideline:

If you create a wrapped generic type, also create a set of "lifts" for clients to use with it

But I'm not going explain how right now!

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

$$1 \times 2 = 3$$

Another one of those things

Some things

A way of combining them

$$\max(1,2) = 2$$

Some things

Another one of those things

A way of combining
them

$$\text{"a"} + \text{"b"} = \text{"ab"}$$

Some things

Another one of
those things

A way of combining them

$$\text{concat}([a],[b]) = [a; b]$$

Some things

Another one of those things

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

$$\overbrace{1 + 2}^{\text{Is an integer}}$$

$$\overbrace{1 + 2 + 3}^{\text{Is an integer}}$$

$$1 + 2 + 3 + 4$$

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.

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

$$3 + 7$$

- **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 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} ]

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

orderLines |> List.reduce addLine
```

Any combination of monoids is also a monoid

Pattern:
Convert non-monoids to monoids

Not a monoid

A monoid

Customer → Customer Stats

\+ \+

Customer $\xrightarrow{\text{Map}}$ Customer Stats

\+ \+

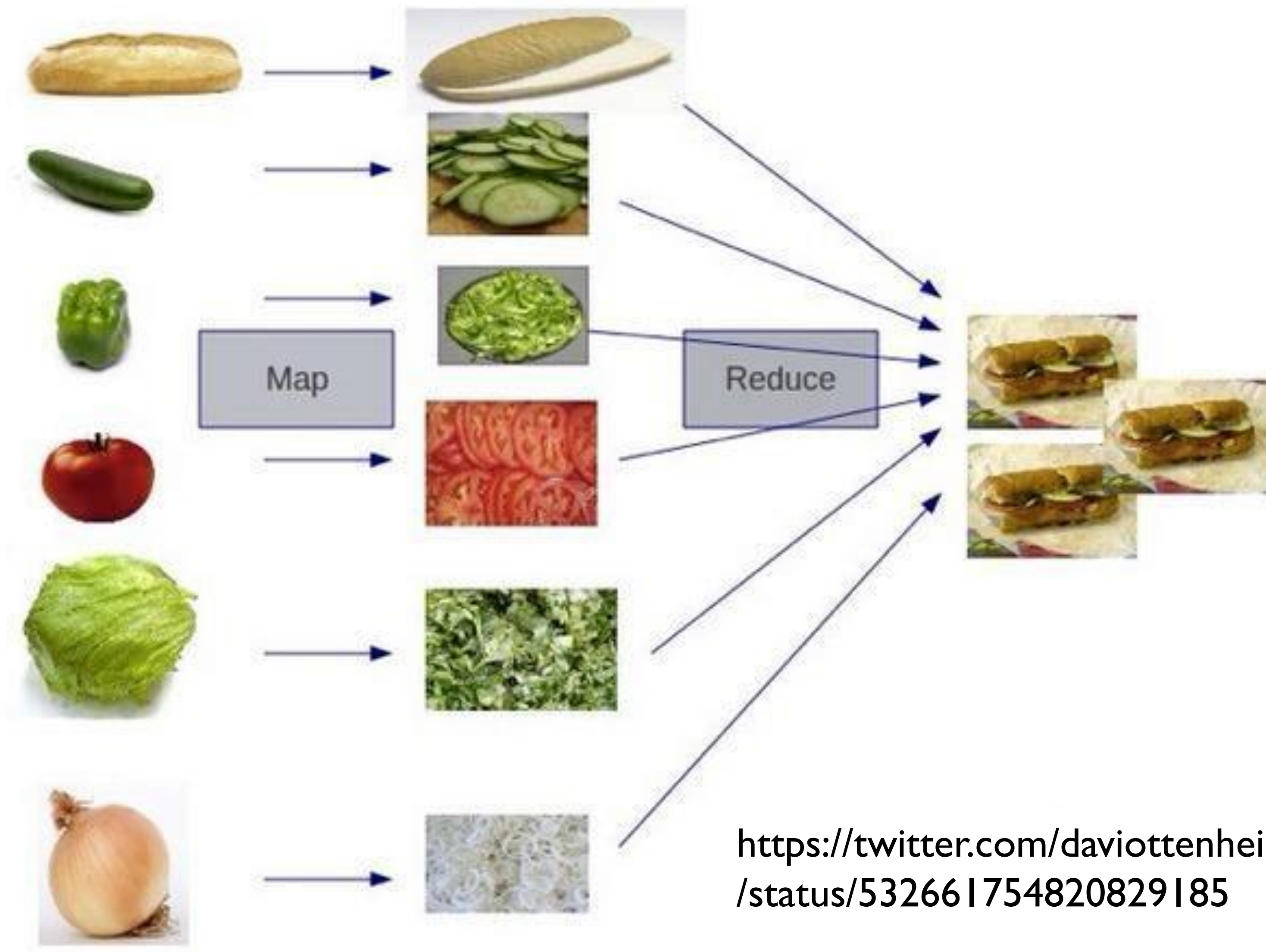
Customer → Customer Stats

Reduce

Summary Stats

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) $\longrightarrow$ Summary (Mon)

\+ \+

Log File (Tue) $\xrightarrow{\textit{Map}}$ Summary (Tue)

\+ \+

Log File (Wed) $\longrightarrow$ Summary (Wed)

=

Really big file

Much more efficient for incremental updates

"Monoid homomorphism"

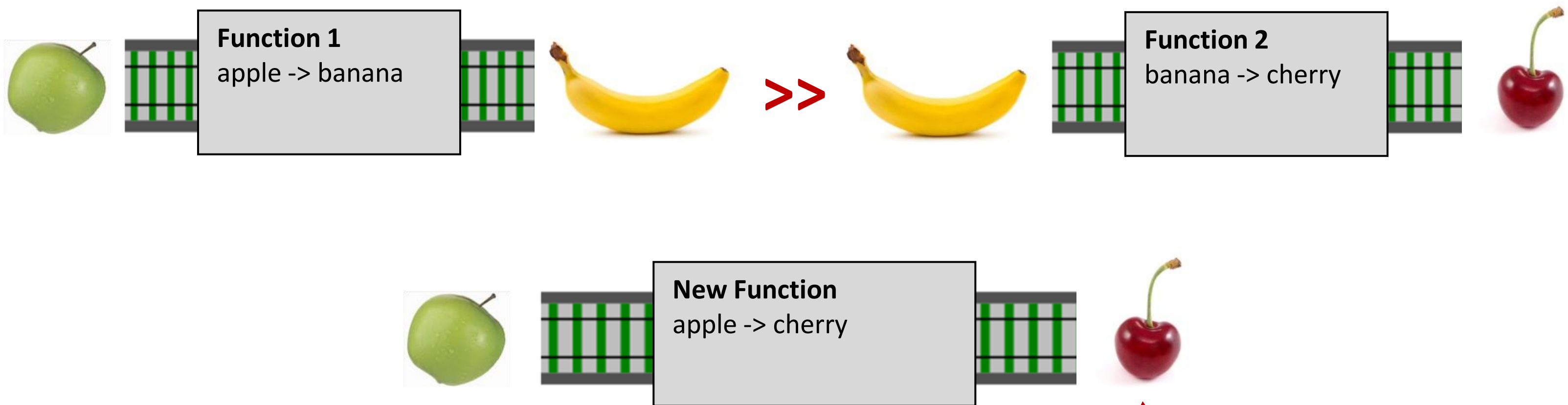
Pattern:
Seeing monoids everywhere

Metrics guideline:
Use counters rather than rates

Alternative metrics guideline:
Make sure your metrics are monoids

- incremental updates
- can handle missing data

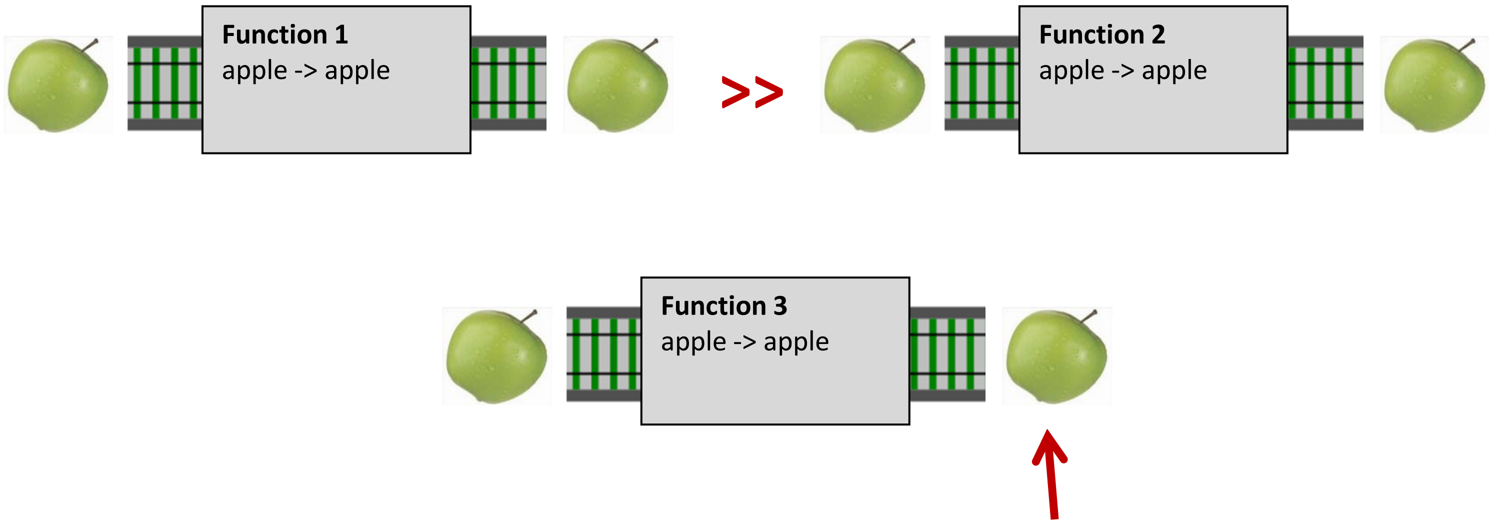
Is function composition a monoid?



Not the same thing.

Not a monoid ☹

Is function composition a monoid?



Same thing

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

Endomorphism example

Endomorphisms

```
let plus1 x = x + 1           // int->int
let times2 x = x * 2          // int->int
let subtract42 x = x - 42     // int->int

let functions = [
    plus1
    times2
    subtract42 ]
```

Put them in a list

```
let newFunction =             // int->int
    functions |> List.reduce (>>)
```

Reduce them

Another endomorphism

```
newFunction 20  // => 0
```

Event sourcing

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

For example: Event sourcing

$$\text{Event} \;\texttt{->}\; \overbrace{\text{State} \;\texttt{->}\; \text{State}}^{\text{Is an endomorphism}}$$

Event sourcing example

Partial application of event

```
let applyFns = [
    apply event1               // State -> State
    apply event2               // State -> State
    apply event3]              // State -> State

let applyAll =                 // State -> State
    applyFns |> List.reduce (>>)

let newState = applyAll oldState
```

A function that can apply all the events in one step

- incremental updates
- can handle missing events

Bind

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

For example: Option.Bind

(fn param) -> $\overbrace{\text{Option -> Option}}^{\text{Is an endomorphism}}$

Event sourcing example

Partial application of Bind

```
let bindFns = [
    Option.bind (fun x->
        if x > 1 then Some (x*2) else None)
    Option.bind (fun x->
        if x < 10 then Some x else None)
    ]

let bindAll =                      // Option->Option
    bindFns |> List.reduce (>>)

Some 4 |> bindAll    // Some 8
Some 5 |> bindAll    // None
```

Predicates as monoids

Not an endomorphism

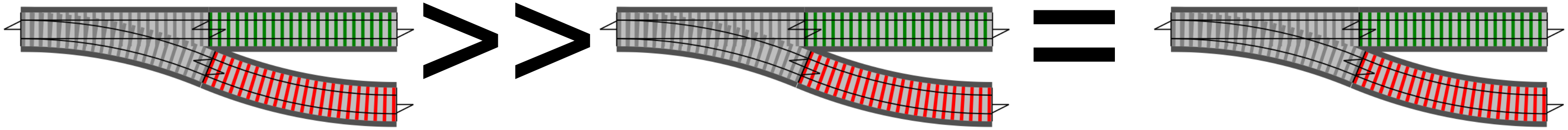
```
type Predicate<'A> = 'A -> bool

let predAnd pred1 pred2 x =
    if pred1 x
    then pred2 x
    else false

let predicates = [
    isMoreThan10Chars        // string -> bool
    isMixedCase              // string -> bool
    isNotDictionaryWord      // string -> bool
    ]
```

But can still be combined

```
let combinedPredicate =     // string -> bool
    predicates |> List.reduce (predAnd)
```

Pattern:
Monads are monoids

Series combination

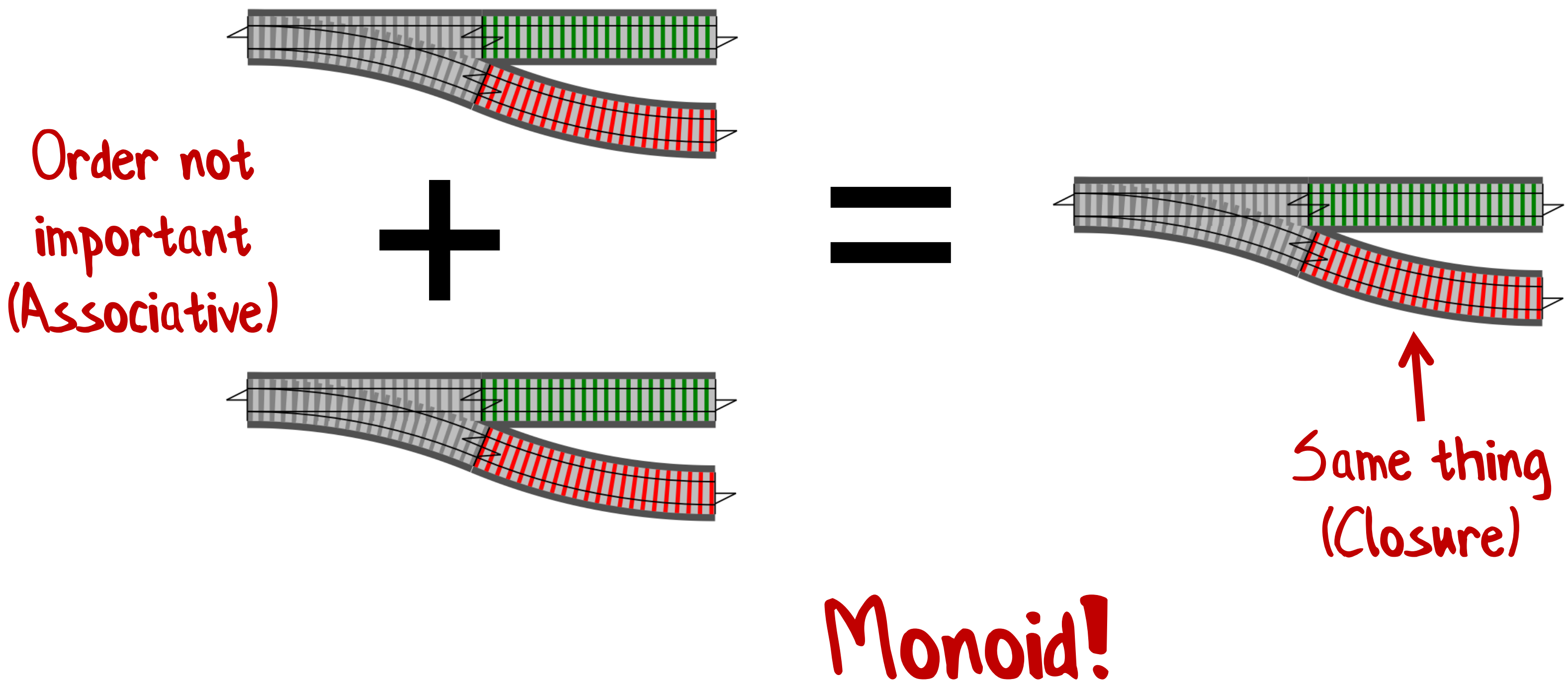


Order not important (Associative)

Result is same kind of thing (Closure)

Monoid!

Parallel combination



Monad laws

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

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

fsharp.org

THANKS!

More F#
here!