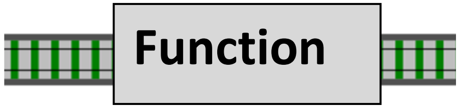
THE PRINCIPLES OF
FUNCTIONAL PROGRAMMING

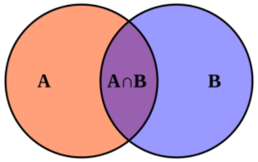
Core principles of FP

Functions are things

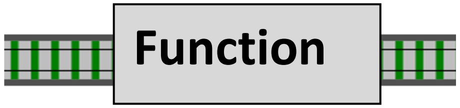


Composition everywhere

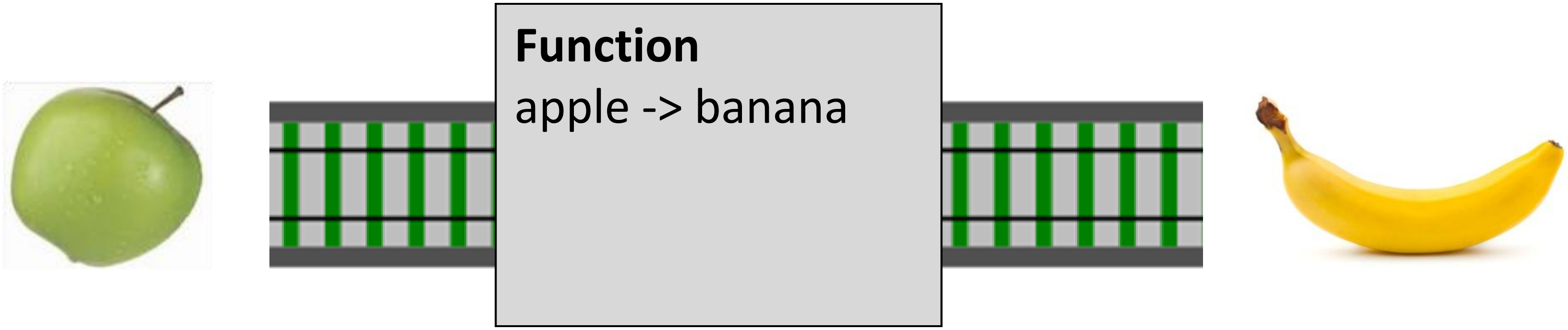
Types are not classes



Core FP principle: Functions are things



Functions as things



Function

apple -> banana

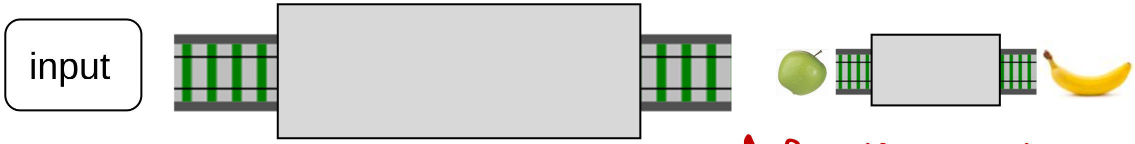
A function is a thing which transforms inputs to outputs

Another word for reusable!

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

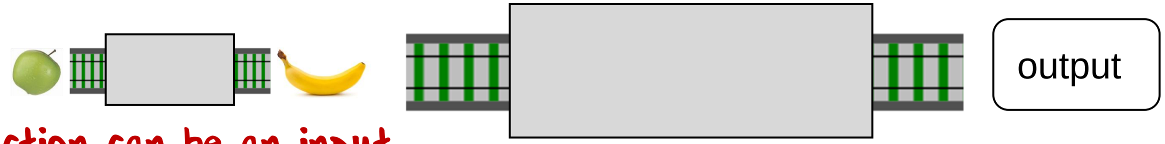
It can be used for inputs and outputs of other functions

A function is a standalone thing



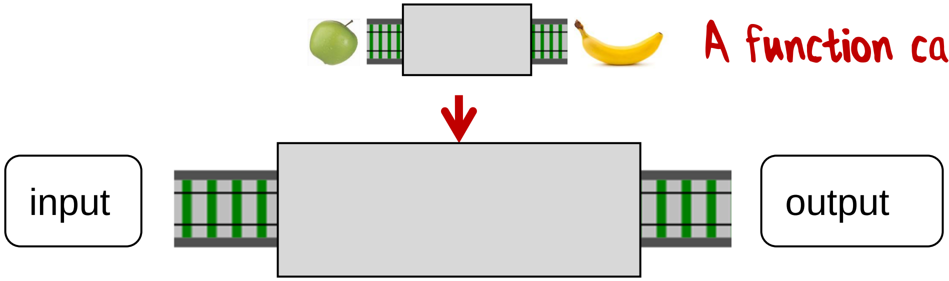
A function can be an output

A function is a standalone thing



A function can be an input

A function is a standalone thing



input

output

A function can be a parameter

You can build very complex systems from this simple foundation!

Core FP principle:
Composition everywhere

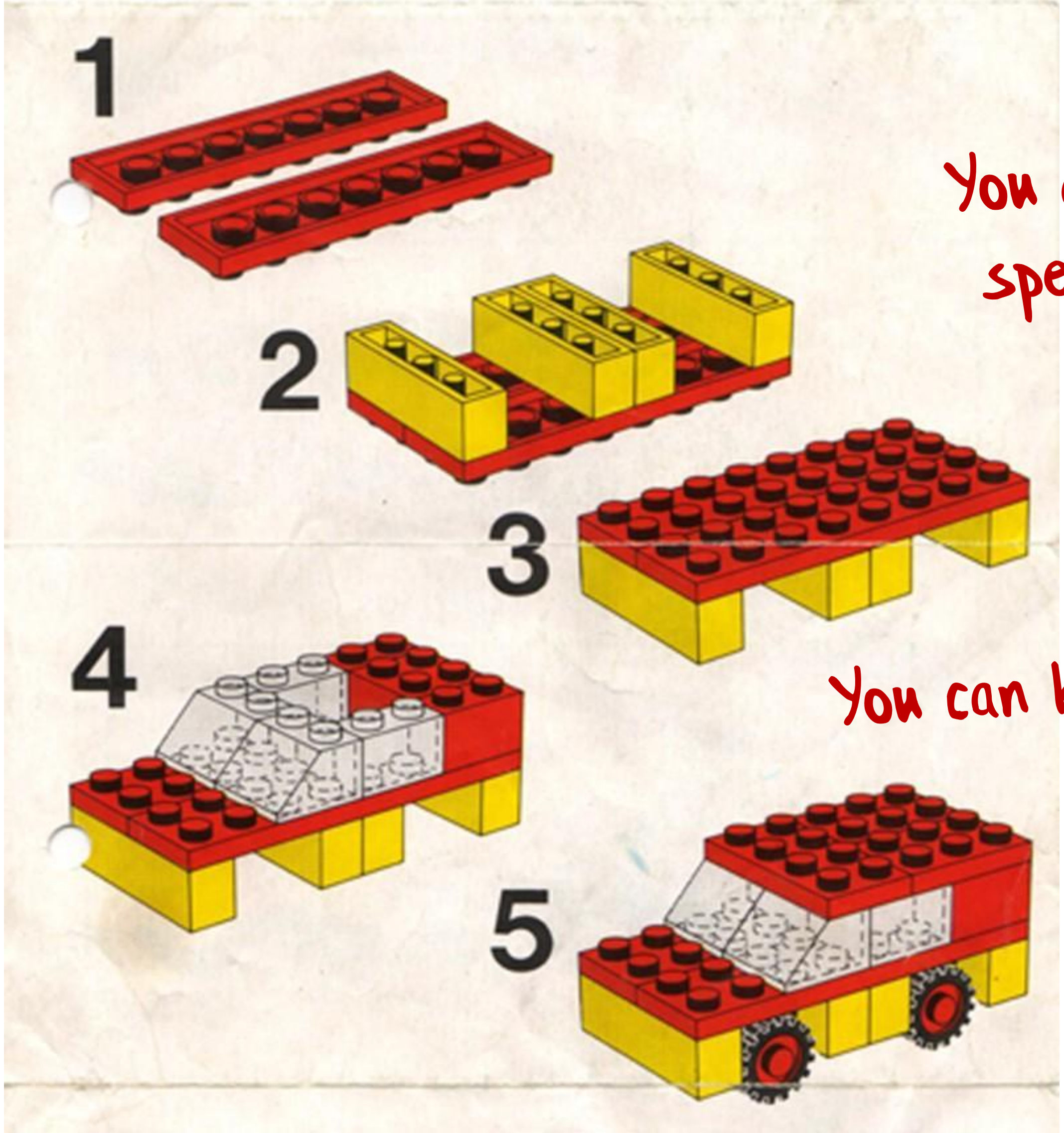
What is Composition?

Lego Philosophy

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

All pieces are designed to be connected

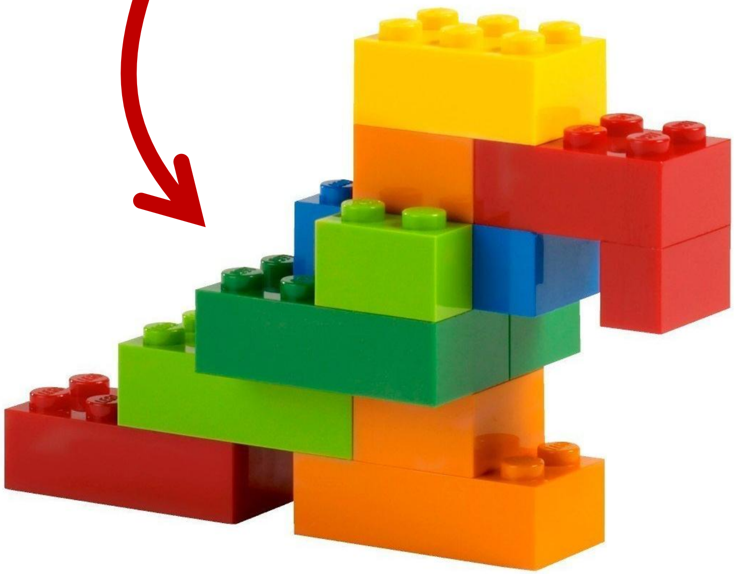
Connect two pieces together and get another "piece" that can still be connected

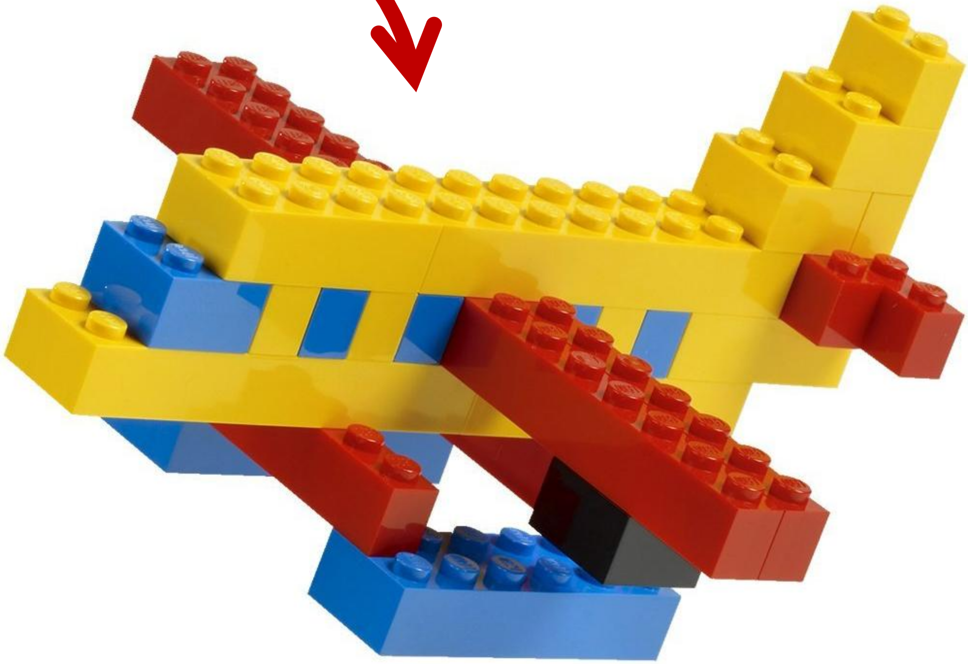


You don't need to create a special adapter to make connections.

You can keep adding and adding.

The pieces are reusable in different contexts

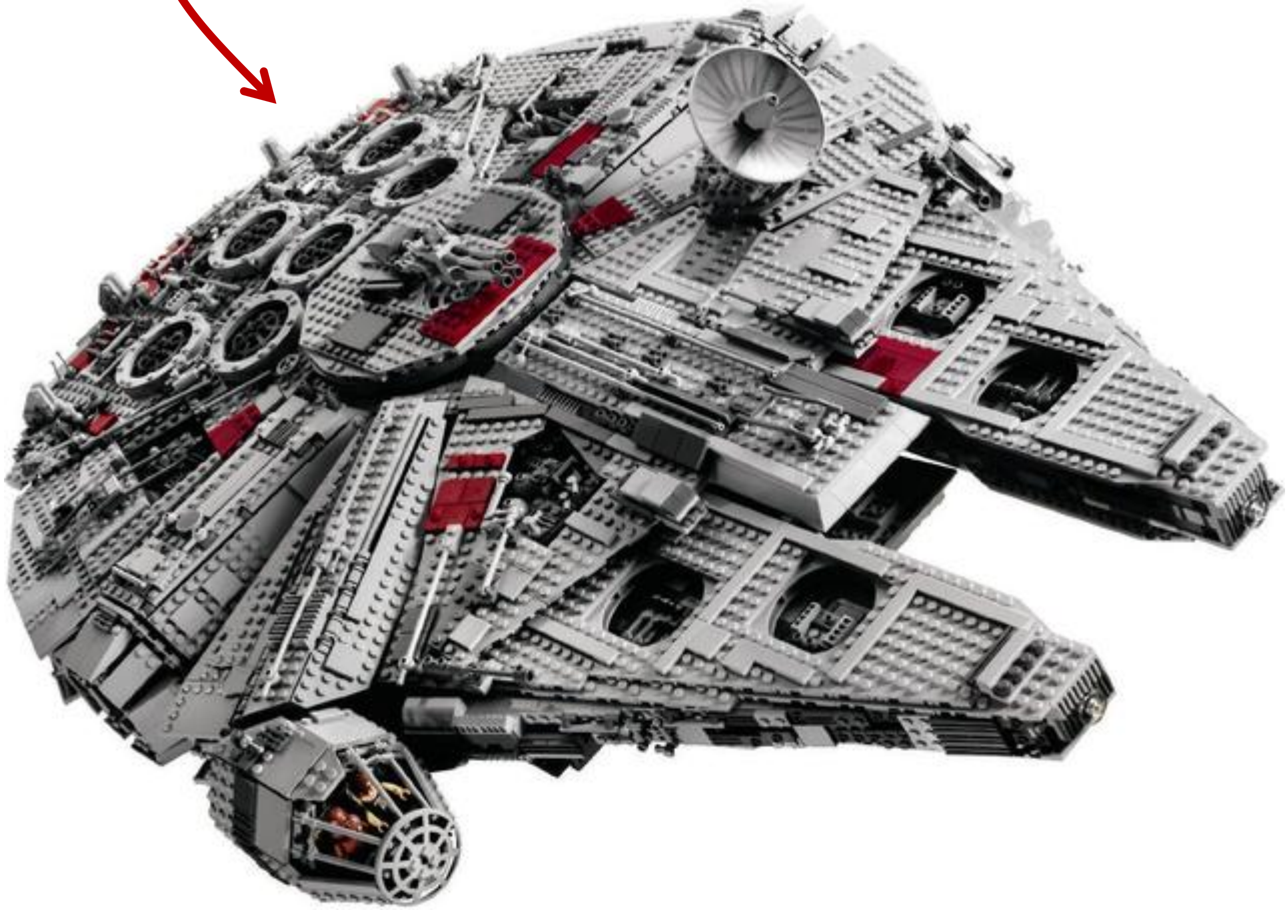




They are self contained.
No strings attached (literally).

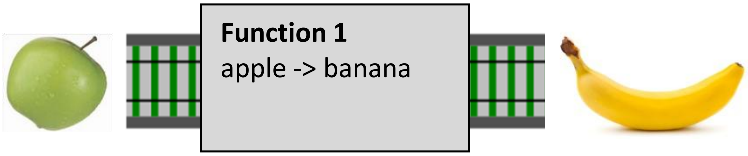
Make big things from small things in the same way

The Power of Composition



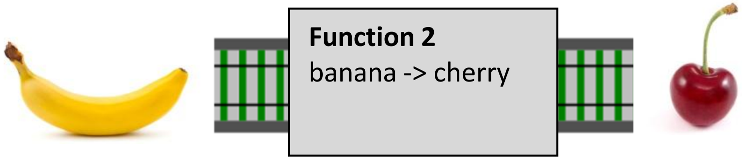
Function Composition

Function composition



Function 1

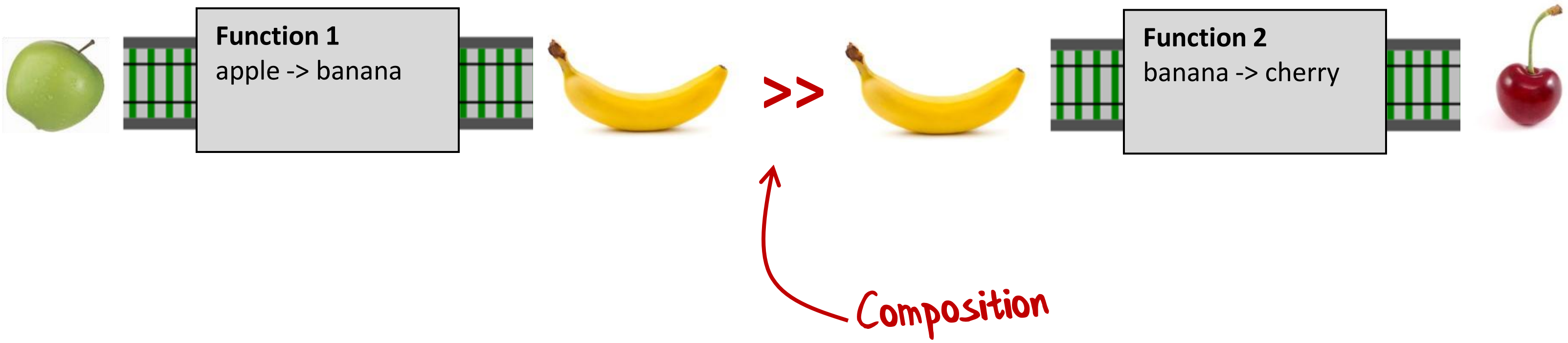
apple -> banana



Function 2

banana -> cherry

Function composition



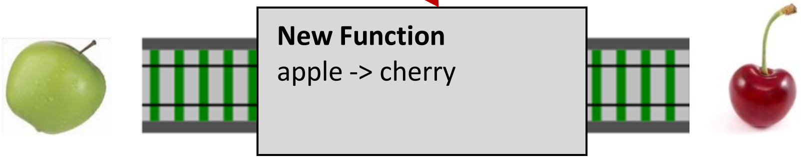
Function 1
apple -> banana

>>

Function 2
banana -> cherry

Composition

Function composition

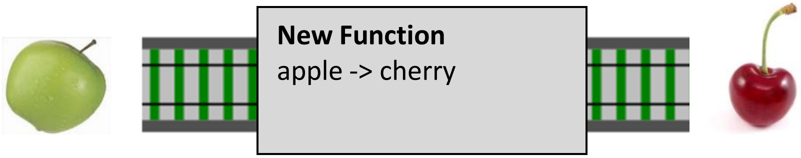


New function

Can't tell it was built
from smaller functions!

Where did the banana go?
(abstraction)

Function composition



New Function

apple -> cherry

A Very Important Point: For composition to work properly:

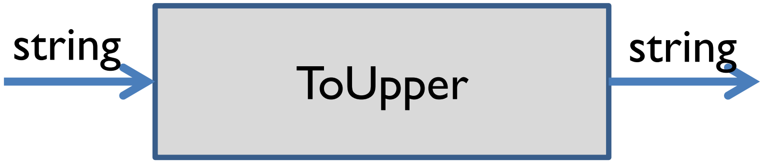
- Data must be immutable
- Functions must be self-contained, with no strings attached:

  no side-effects, no I/O, no globals, etc

Building big things from functions

It's compositions all the way up

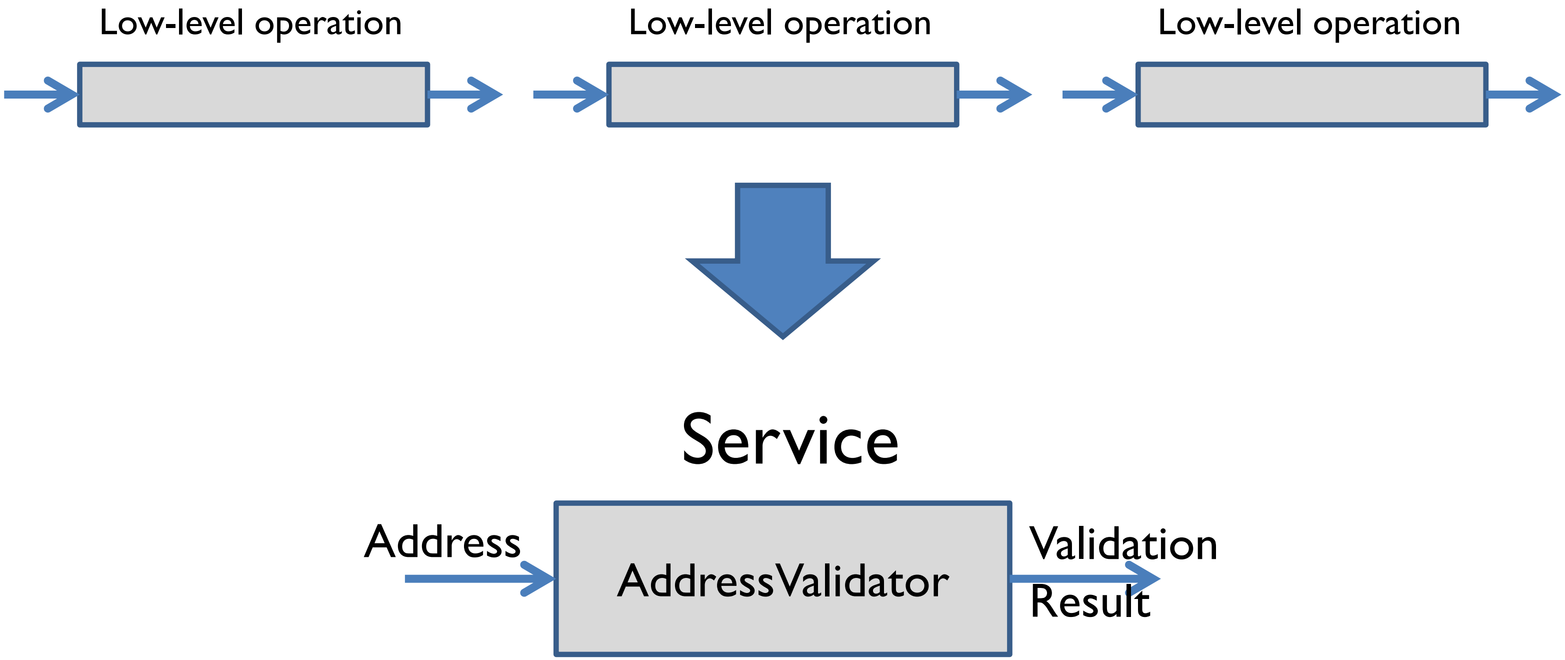
Low-level operation



string

ToUpper

string



Low-level operation

Low-level operation

Low-level operation

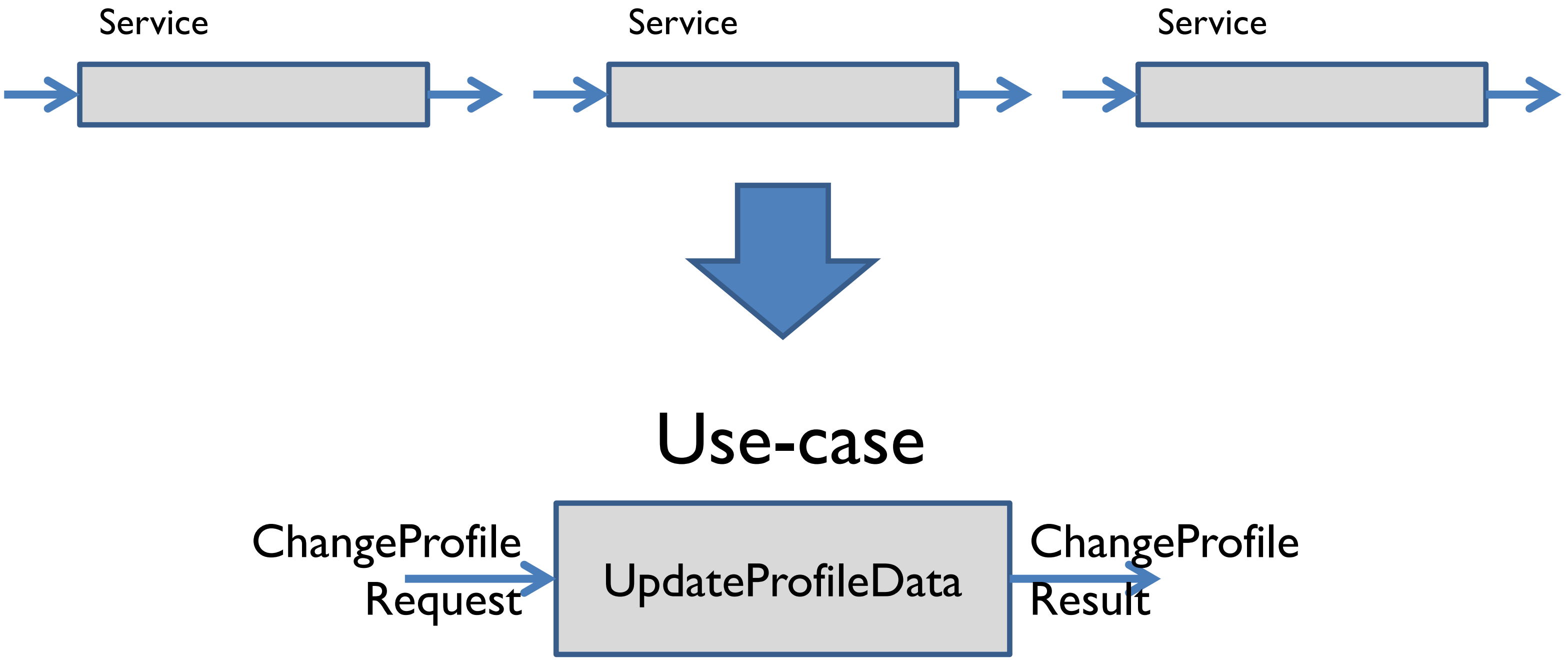
Service

Address

AddressValidator

Validation Result

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



Service

Service

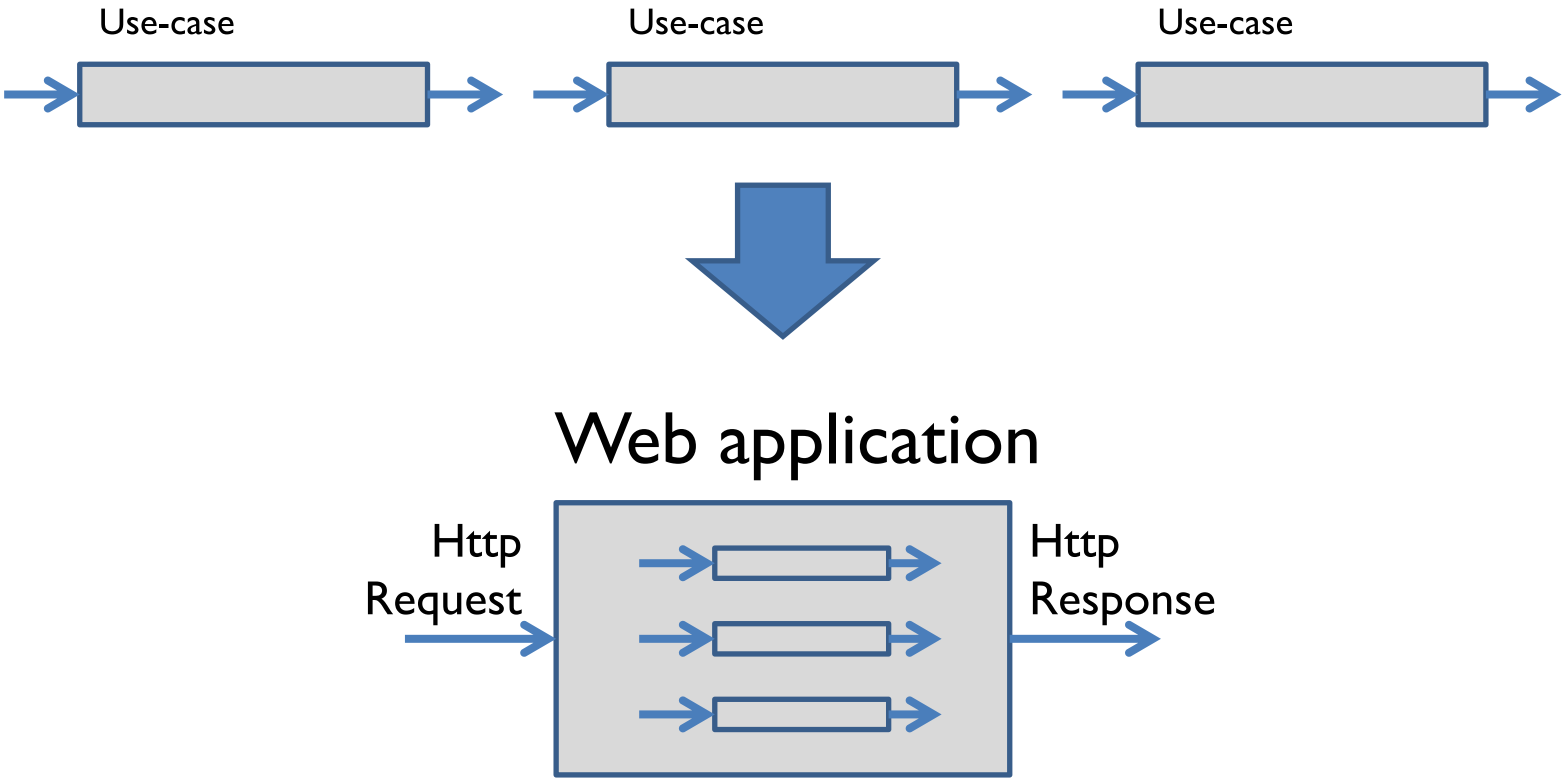
Service

Use-case

ChangeProfile Request

UpdateProfileData

ChangeProfile Result



Use-case

Use-case

Use-case

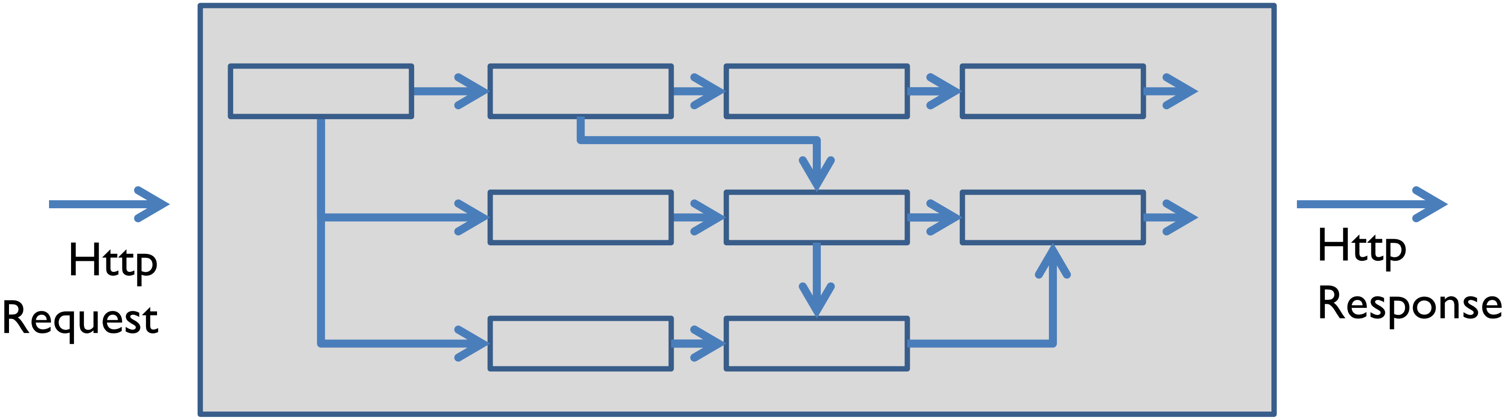
Web application

Http Request

Http Response

"Composition is fractal"

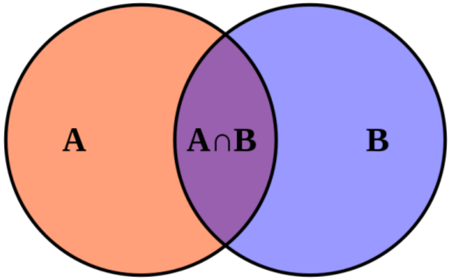
The Power of Composition



Http Request

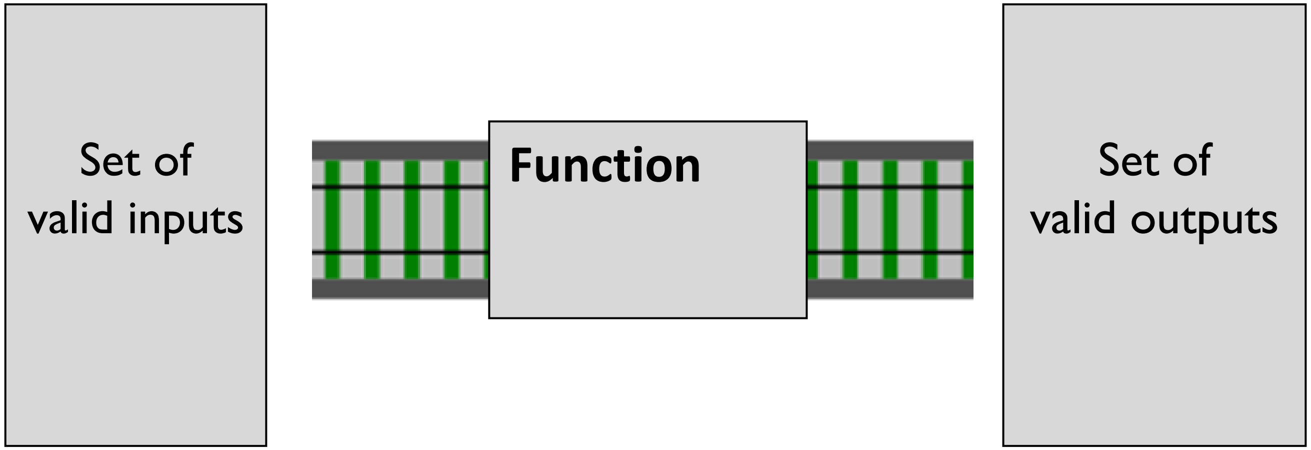
Http Response

Core FP principle:
Types are not classes



So, what is a type then?

A type is a just a name
for a set of things

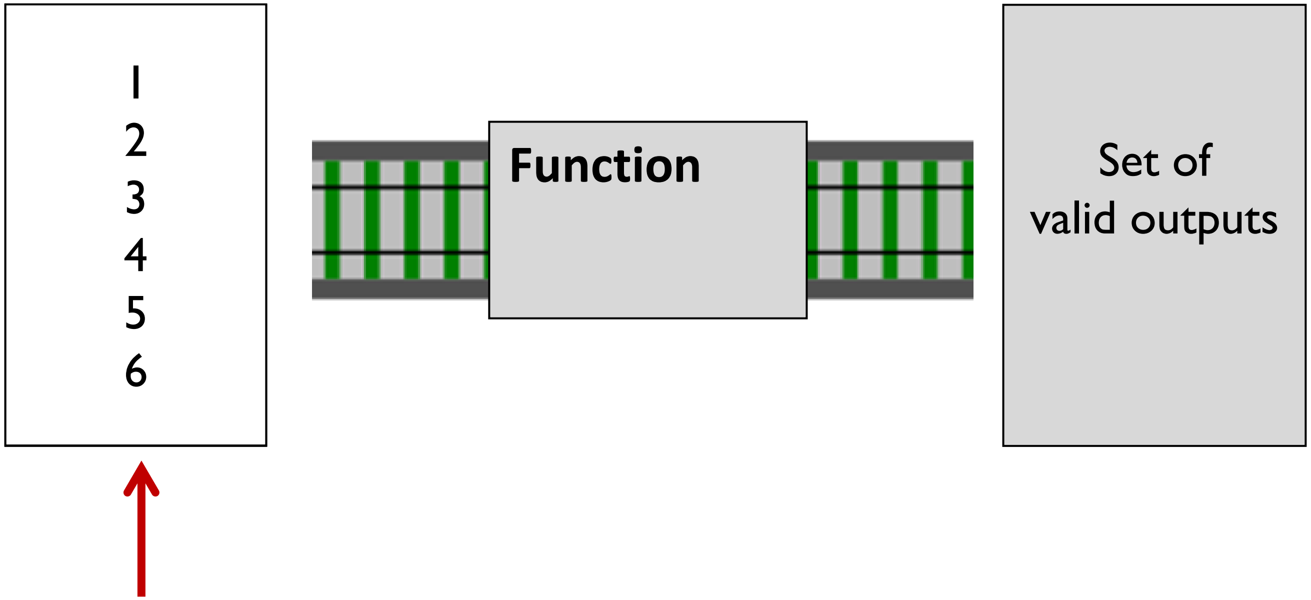


Set of
valid inputs

Function

Set of
valid outputs

A type is a just a name for a set of things



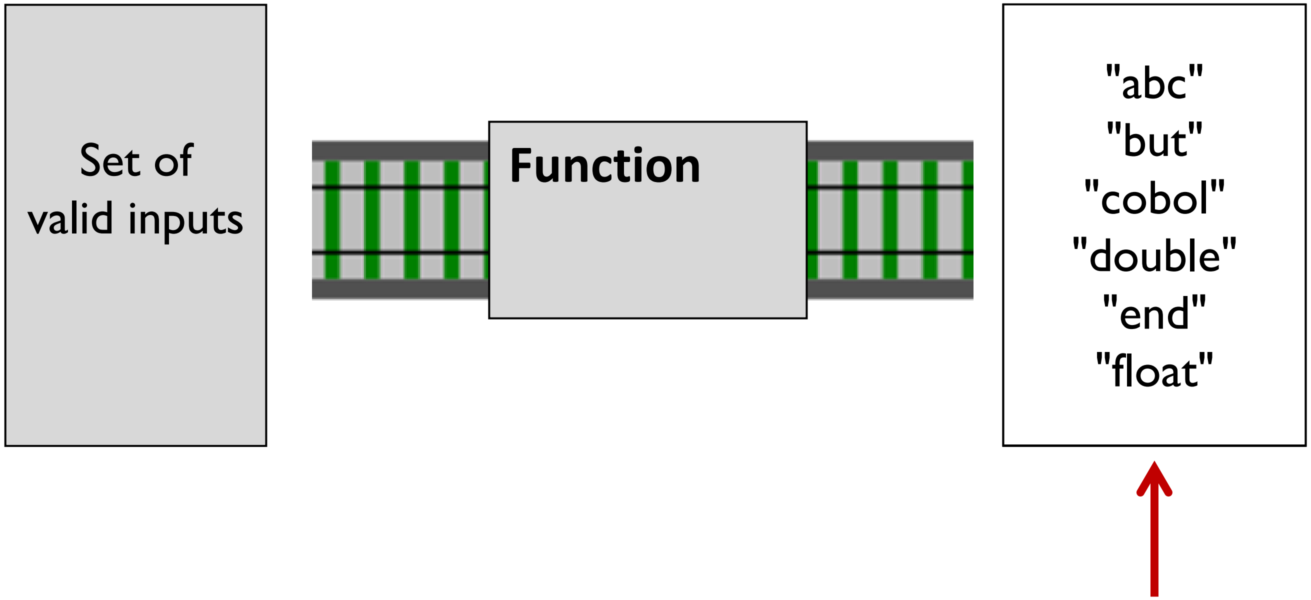
1
2
3
4
5
6

Function

Set of valid outputs

This is type "integer"

A type is a just a name for a set of things



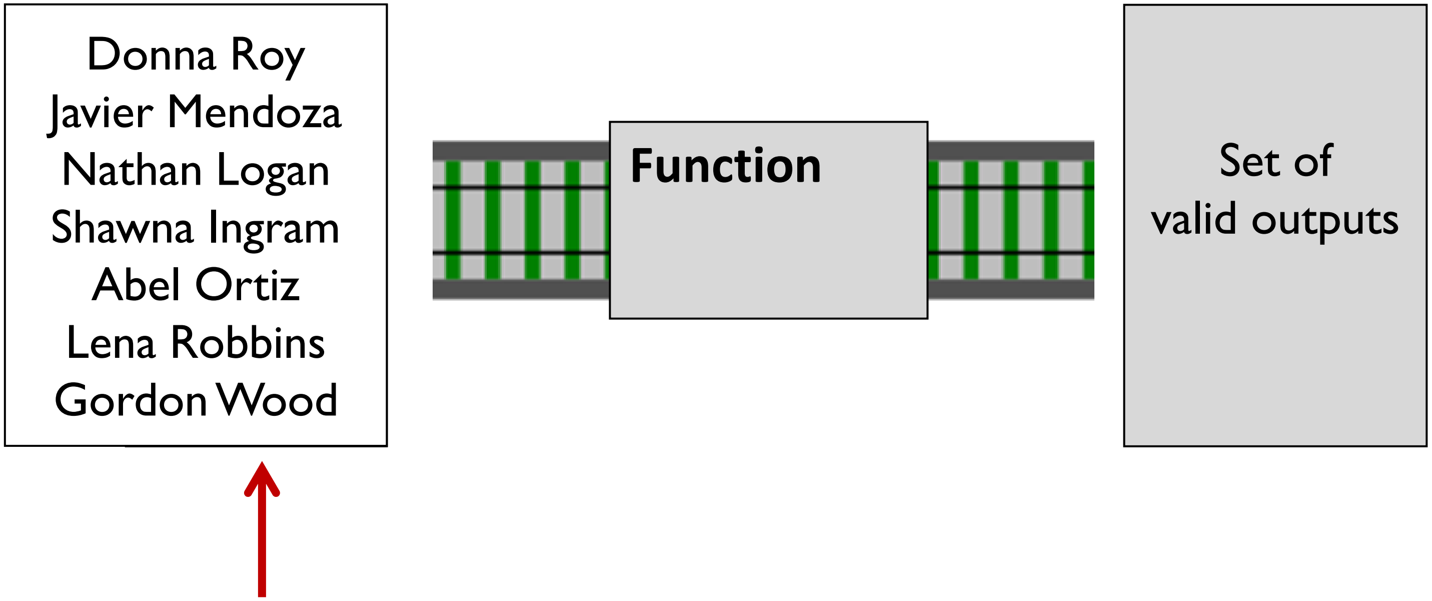
Set of valid inputs

Function

"abc"
"but"
"cobol"
"double"
"end"
"float"

This is type "string"

A type is a just a name for a set of things



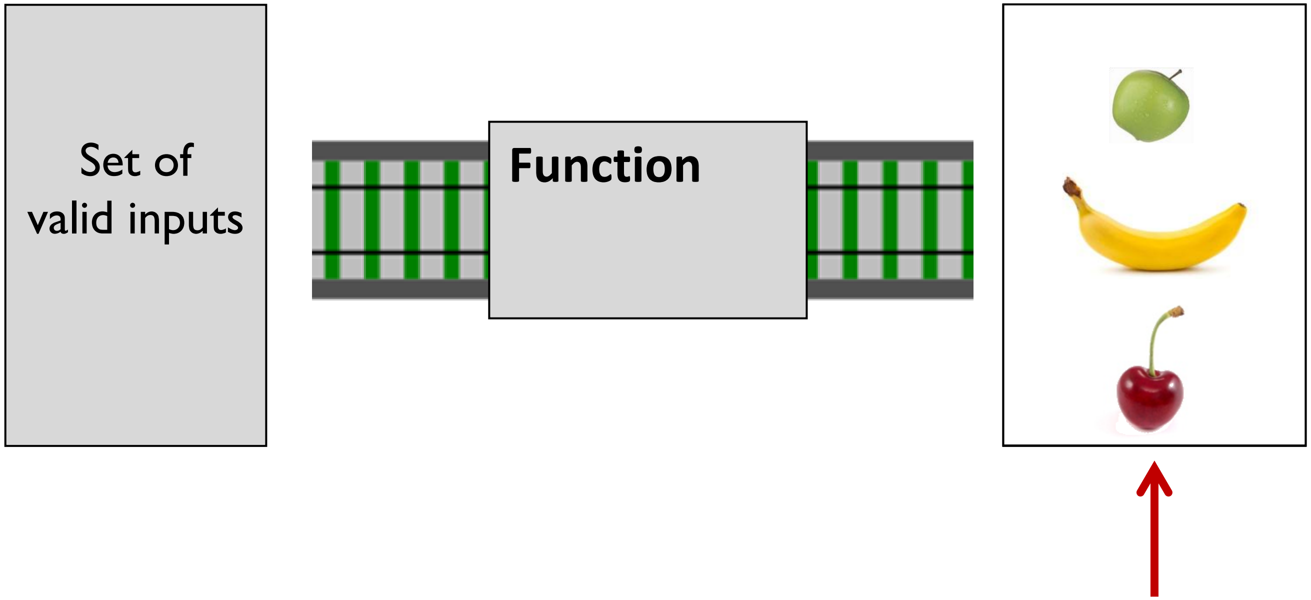
Donna Roy
Javier Mendoza
Nathan Logan
Shawna Ingram
Abel Ortiz
Lena Robbins
Gordon Wood

Function

Set of
valid outputs

This is type
"Person"

A type is a just a name for a set of things

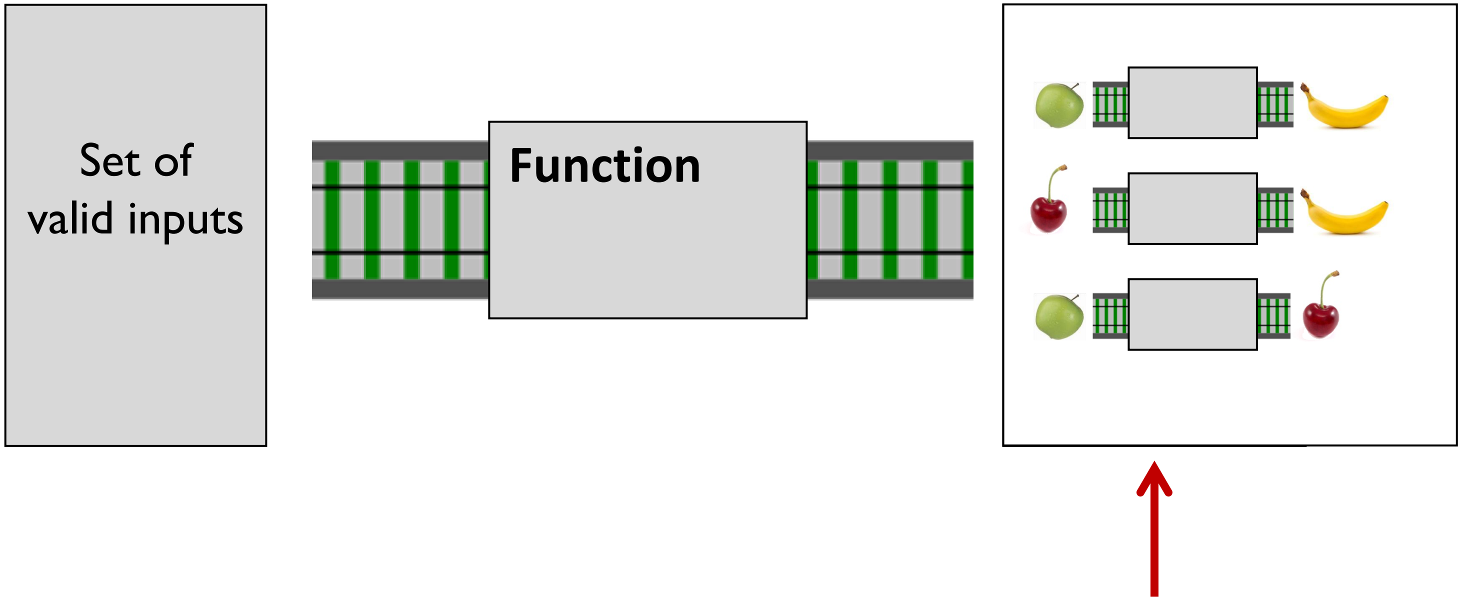


Set of valid inputs

Function

This is type "Fruit"

A type is a just a name for a set of things

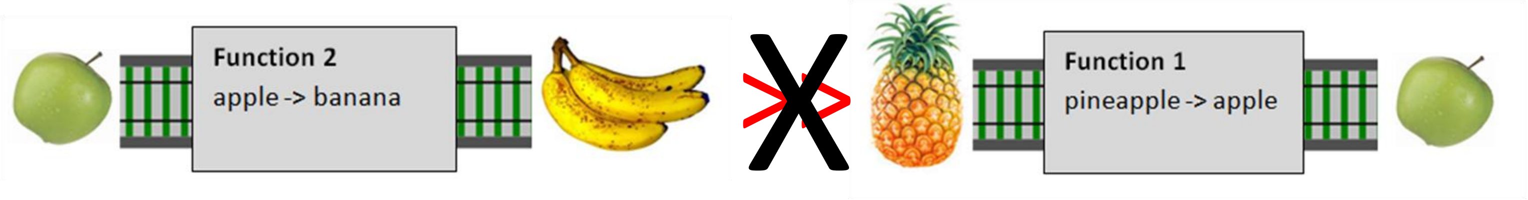


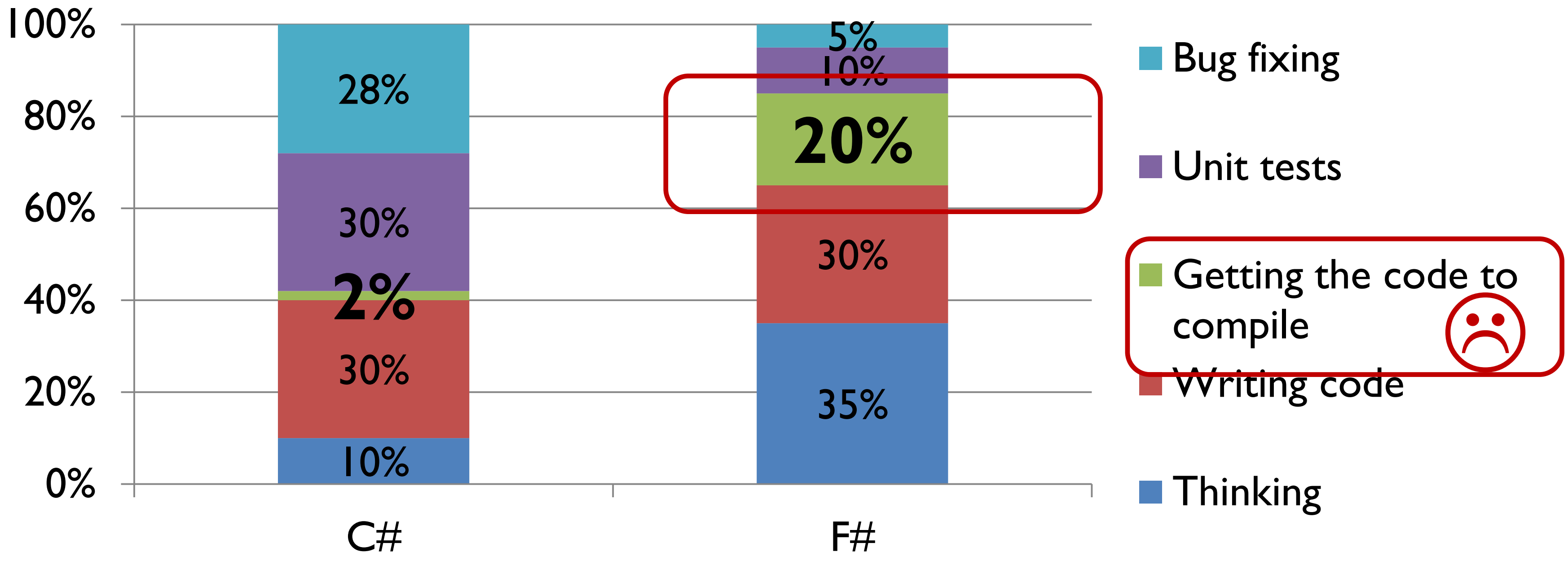
Set of valid inputs

Function

This is a type of Fruit->Fruit functions

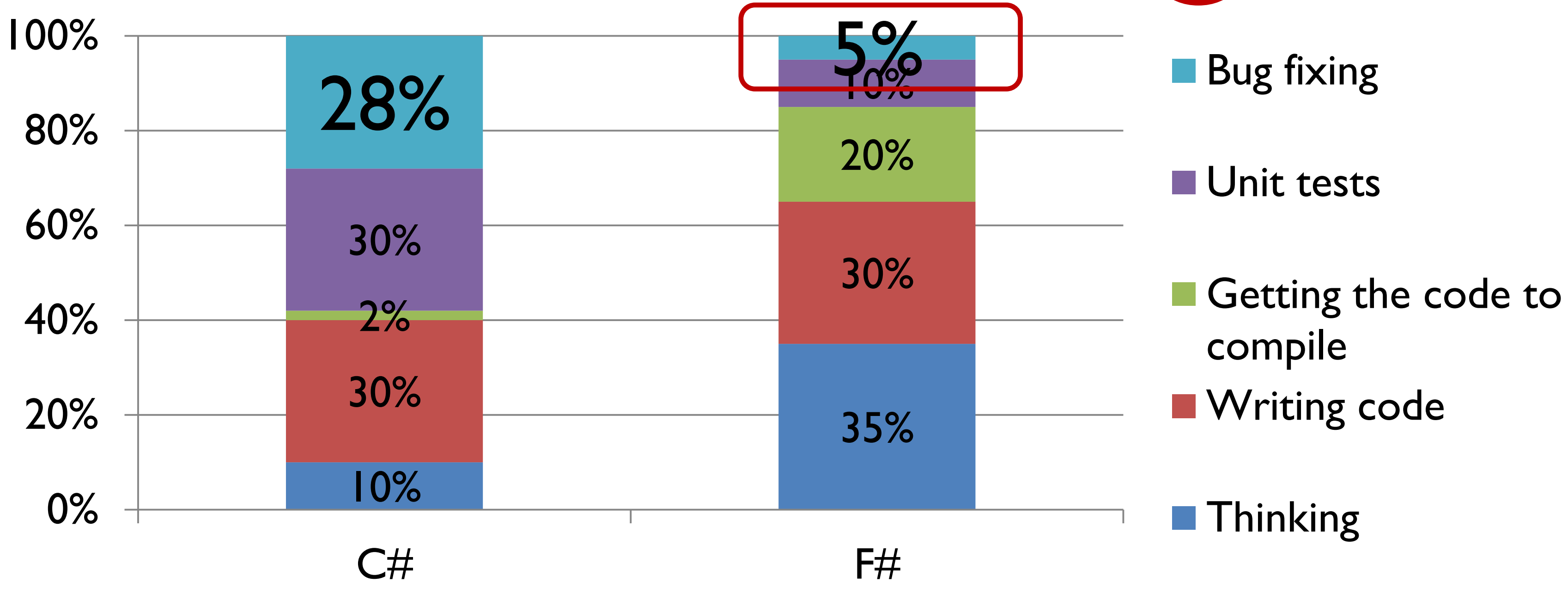
Composition is type checked!





But the good news is...

A lot less bug fixing! ☺



Composition everywhere:
Types can be composed too

Composable

~~Algebraic~~ type system

New types are built from smaller types by:

- Composing with “AND”
- Composing with “OR”

Only possible because behavior is separate from data!

Compose with "AND"

FruitSalad = One each of 🍏 and 🍌 and 🍒

Example: pairs, tuples, records

A record type

```
type FruitSalad = {
   Apple: AppleVariety
   Banana: BananaVariety
   Cherry: CherryVariety
   }
```

Compose with “OR”

Not available in C#

Snack = or or

A choice type

```
    type Snack =
         | Apple of AppleVariety
         | Banana of BananaVariety
         | Cherry of CherryVariety
```

Real world example of type composition

Example of some requirements:

We accept three forms of payment:
Cash, Check, or Card.

For Cash we don't need any extra information
For Checks we need a check number
For Cards we need a card type and card number

How would you implement this?

In OO design you would probably implement it as an interface and a set of subclasses, like this:

```
interface IPaymentMethod
{..}

class Cash() : IPaymentMethod
{..}

class Check(int checkNo): IPaymentMethod
{..}

class Card(string cardType, string cardNo) : IPaymentMethod
{..}
```

In F# you would probably implement by composing types, like this:

```
type CheckNumber = int
type CardNumber = string
```

Primitive types

```
type CheckNumber = ...
type CardNumber = …

type CardType = Visa | Mastercard
type CreditCardInfo = {
   CardType : CardType
   CardNumber : CardNumber
   }
```

Choice type (using OR)

Record type (using AND)

```
type CheckNumber = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...

type PaymentMethod =
  | Cash
  | Check of CheckNumber
  | Card of CreditCardInfo
```

Choice type

```
type CheckNumber = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | Check of CheckNumber
  | Card of CreditCardInfo

type PaymentAmount = decimal
type Currency = EUR | USD
```

Another primitive type

Another choice type

Final type built from many smaller types:

The Power of Composition

```
type CheckNumber = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | Check of CheckNumber
  | Card of CreditCardInfo
type PaymentAmount = decimal
type Currency = EUR | USD

type Payment = {
  Amount :  PaymentAmount
  Currency:  Currency
  Method:  PaymentMethod  }
```

Record type

FP design principle:

Types are executable documentation

Types are executable documentation

The domain on one screen!

```
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 CheckNumber = CheckNumber of int

type PaymentMethod =
  | Cash
  | Check of CheckNumber
  | Card of CardType * CardNumber
```

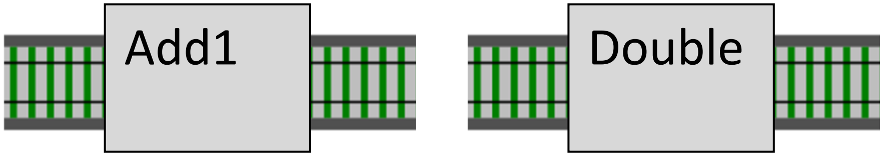
Can you guess what payment methods are accepted?

The End

This is everything you need to know about functional programming

Composition in practice

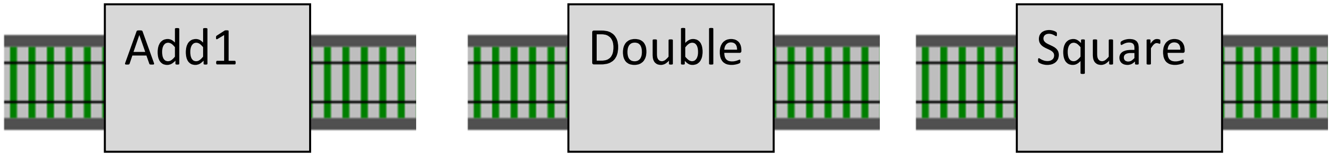
Composition



```
let add1 x = x + 1
let double x = x + x

let add1_double = add1 >> double

let x = add1_double 5    // 12
```

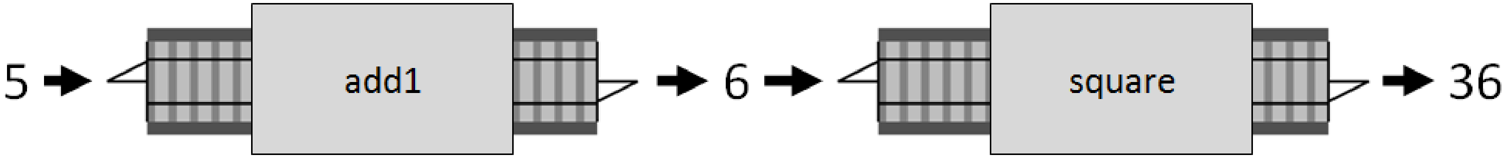
Compositon operator

Composition



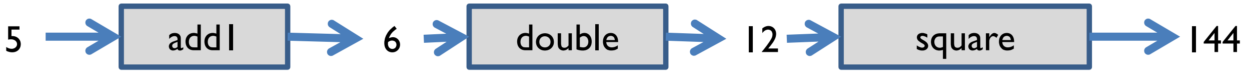
```
let add1_double_square =
   add1 >> double >> square

let x = add1_double_square 5    // 144
```

Piping



```
add1 5                         // = 6
double (add1 5)                // = 12
square (double (add1 5))       // = 144
```

Standard way of nesting function calls
can be confusing if too deep



This is easier to understand

Pipe symbol

```
5 |> add1                         // = 6
5 |> add1 |> double               // = 12
5 |> add1 |> double |> square // = 144
```

pipe

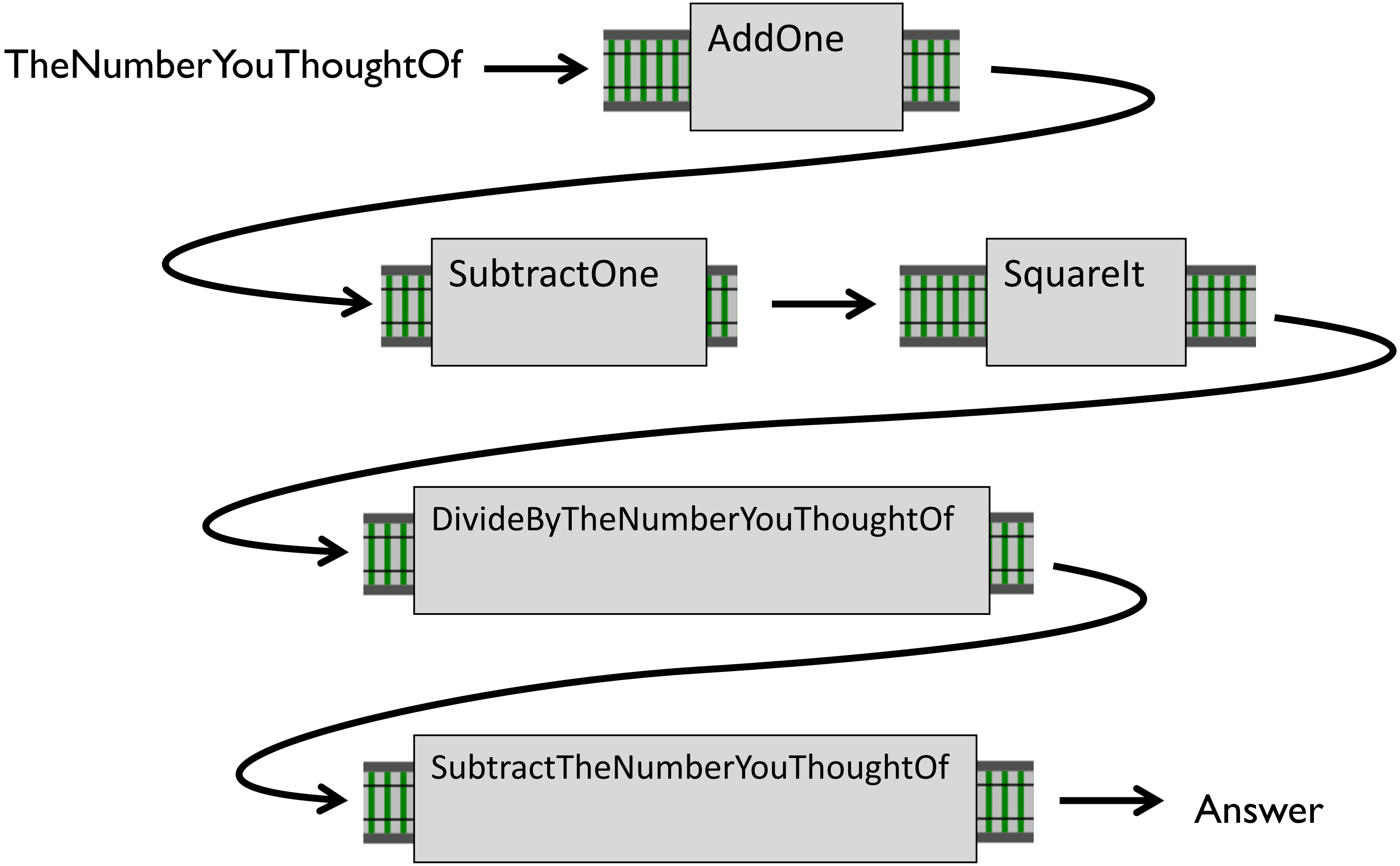
BASIC COMPOSITION:

THINK OF A NUMBER

Think of a number

- Think of a number.
- Add one to it.
- Square it.
- Subtract one.
- Divide by the number you first thought of.
- Subtract the number you first thought of.
- The answer is TWO!

Think of a number



Exercise: Think of a number

```
let thinkOfANumber numberYouThoughtOf =

    // define a function for each step
    let addOne x = x + 1
    let squareIt x = x * x
    let subtractOne x = x - 1
    let divideByTheNumberYouThoughtOf x = x / numberYouThoughtOf
    let subtractTheNumberYouThoughtOf x = x - numberYouThoughtOf

    // then combine them using piping
    numberYouThoughtOf
    |> addOne
    |> squareIt
    |> subtractOne
    |> divideByTheNumberYouThoughtOf
    |> subtractTheNumberYouThoughtOf
```

Exercise:
Play with composable types