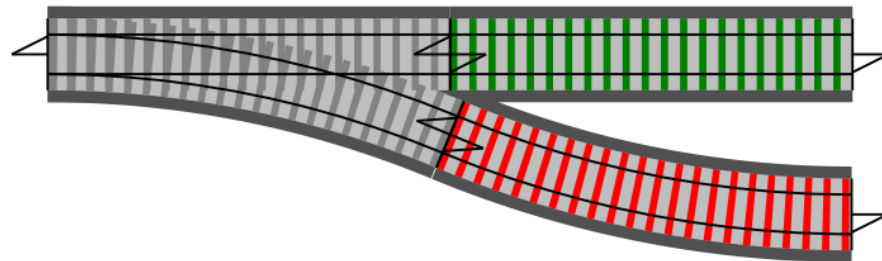


Railway Oriented Programming



What do railways
have to do with
programming?

Scott Wlaschin

<https://github.com/swlaschin/TechTrain2021>

What we will cover today

- Three principles of functional programming
- How to use composition
- Basic error handling in FP
- Building a complete pipeline with errors
- Errors should be part of your domain model

Part I

Three core principles of functional programming

Three principles of FP

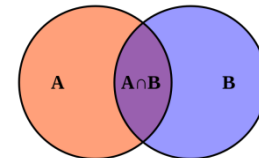
Functions are things



Composition everywhere

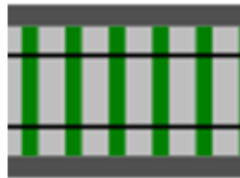


Types are not classes

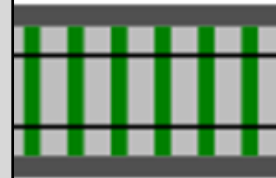


FP principle #1:
Functions are things





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**

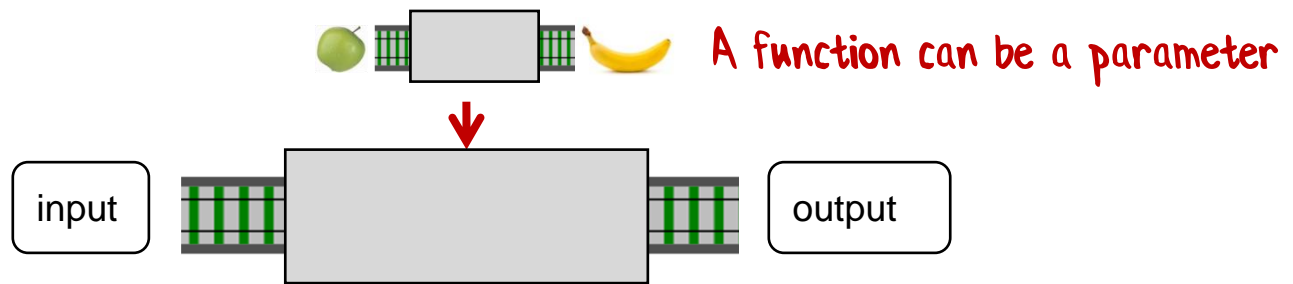
It can be used for inputs and outputs
of other functions



A function can be an output



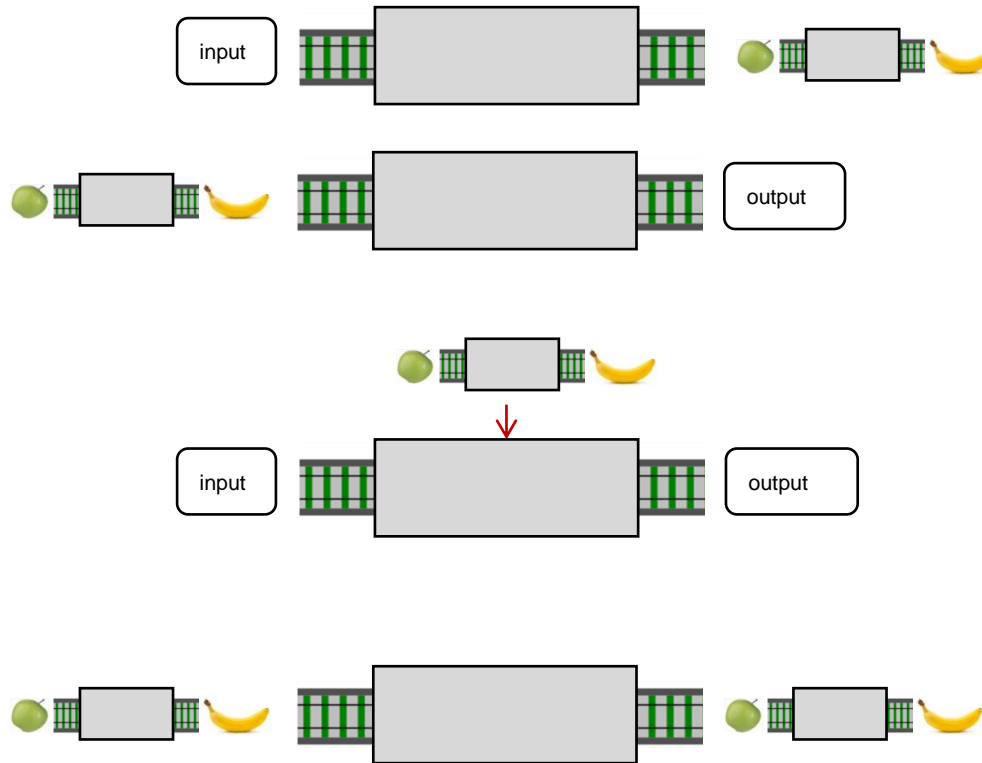
A function can be an input



Similar to the "strategy pattern"



This is a "Function Transformer".
We will be using these a lot!



You can build very
complex systems from
this basic foundation!

FP principle #2:
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

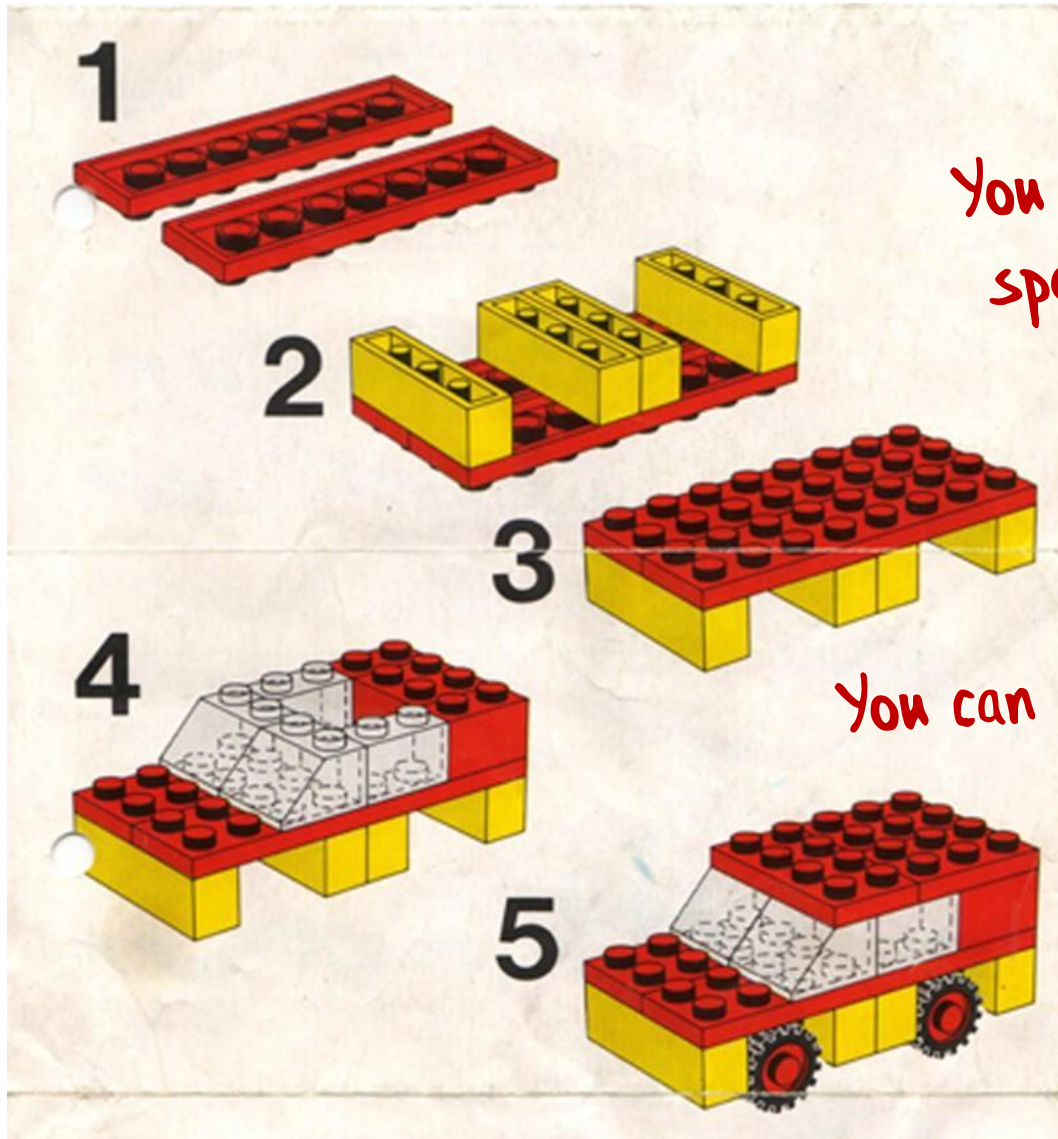
All Lego pieces are designed to be connected



Functional Philosophy

- All functions are designed to be connected

Connect two Lego 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.

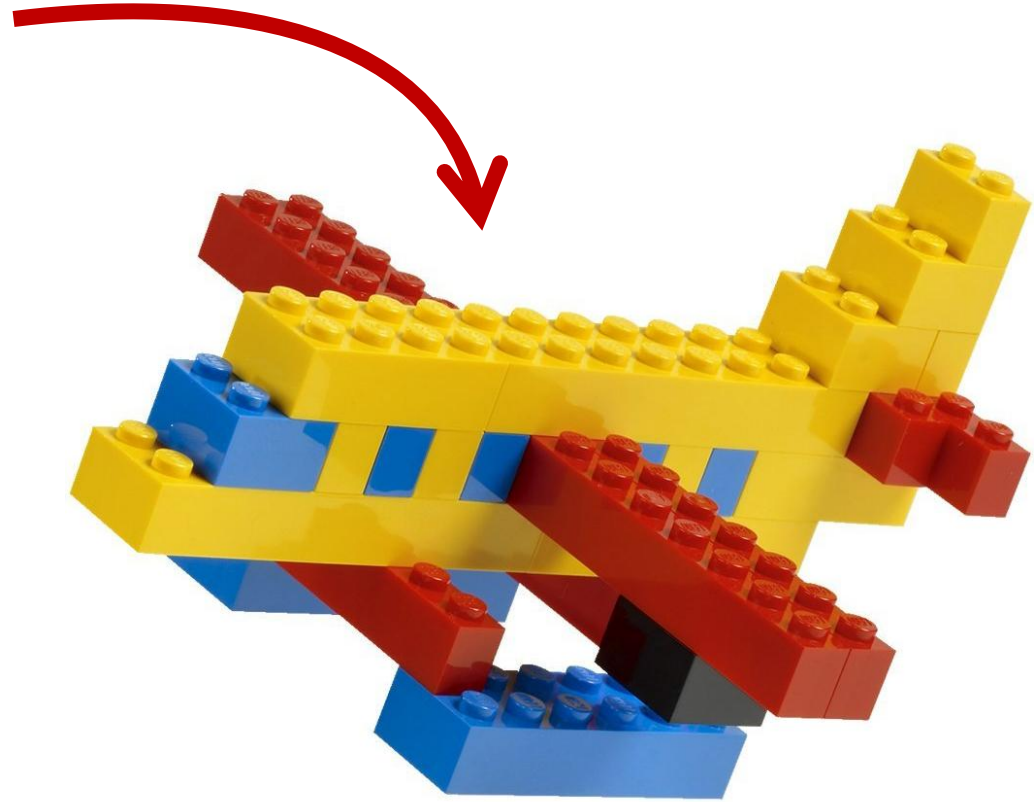
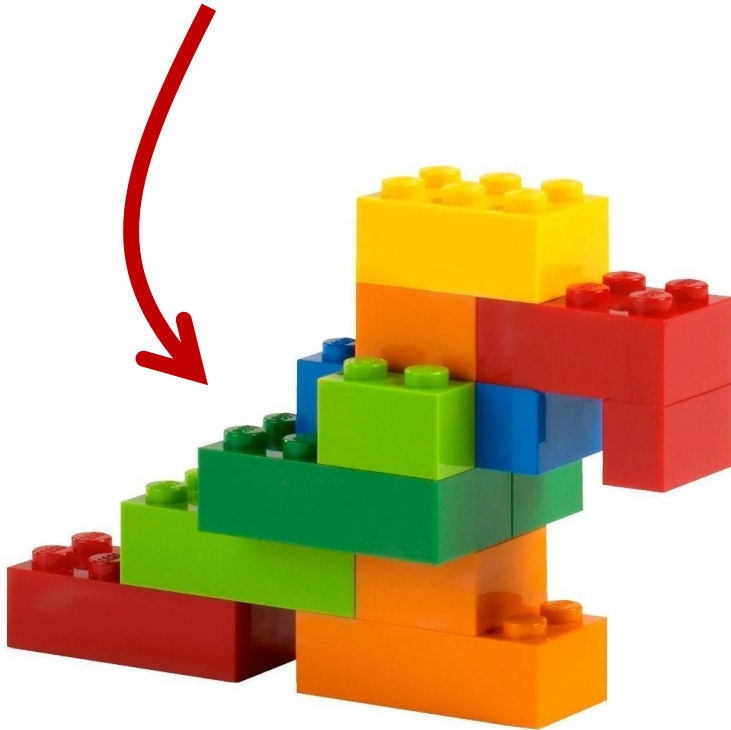
Make big things from small things in the same way



Functional Philosophy

- All functions are designed to be connected
- Connect two functions together and get another function that can still be connected

Lego pieces are reusable in different contexts

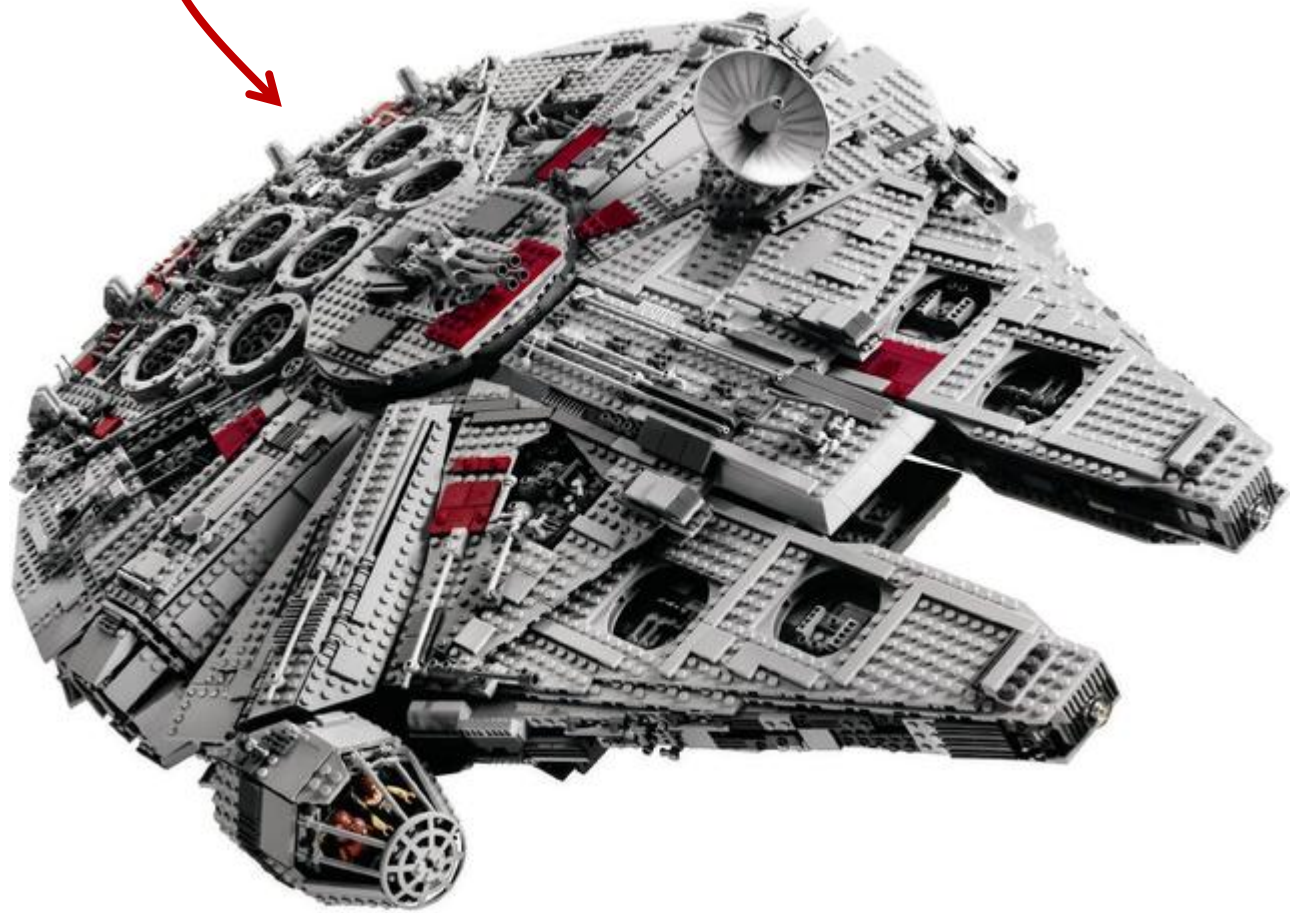


They are self contained.
No strings attached (literally).

Functional Philosophy

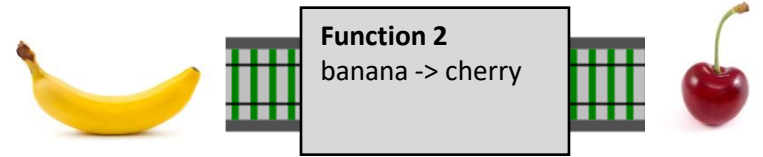
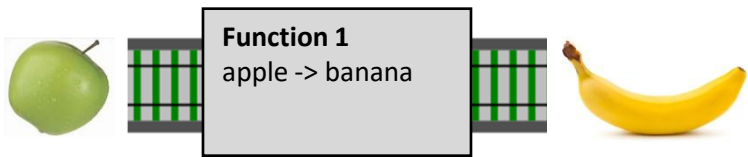
- All functions are designed to be connected
- Connect two functions together and get another function that can still be connected
- The functions are reusable in different contexts

The Power of Composition

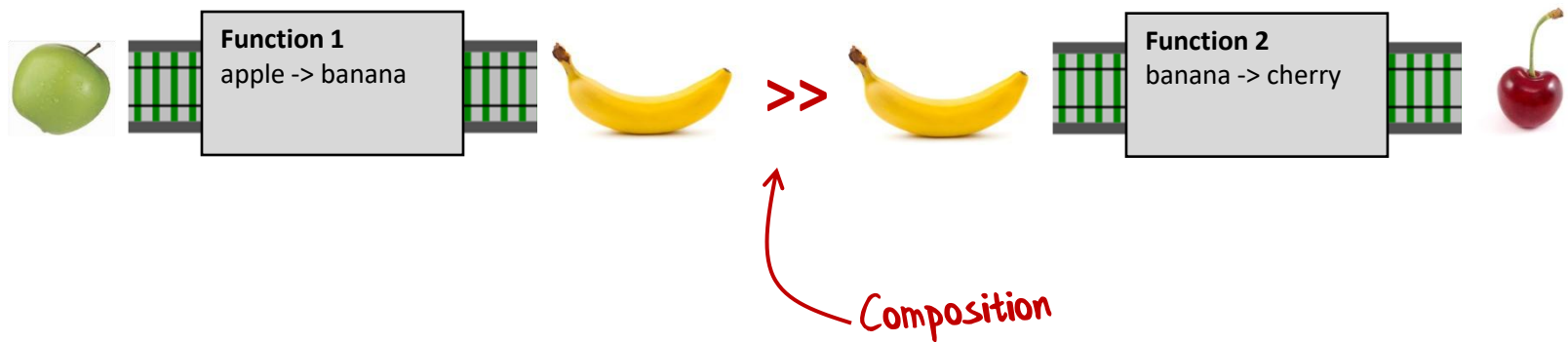


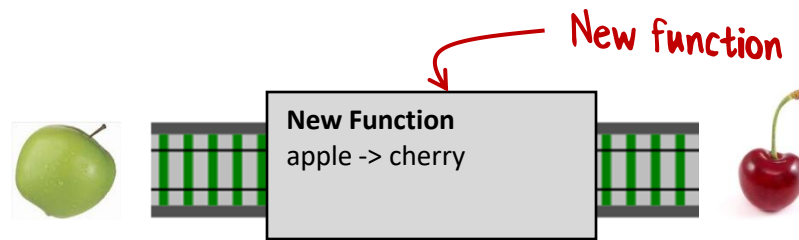
Function Composition





Function composition





Can't tell it was built
from smaller functions!

Where did the banana go?

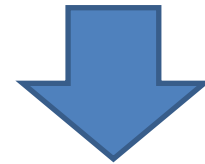
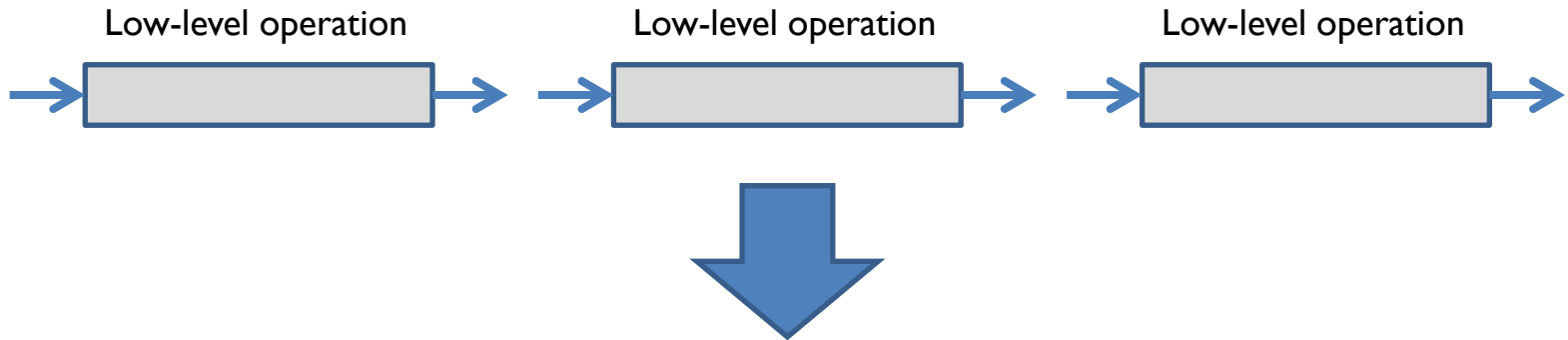
Building big things from functions

It's compositions all the way up



Low-level operation

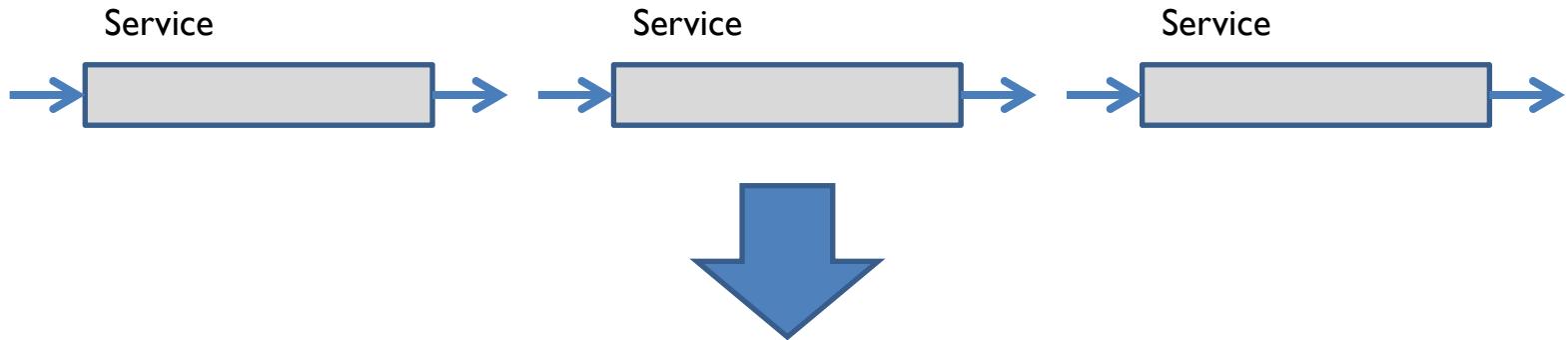




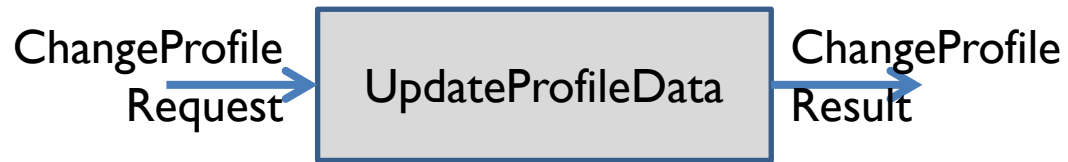
Service

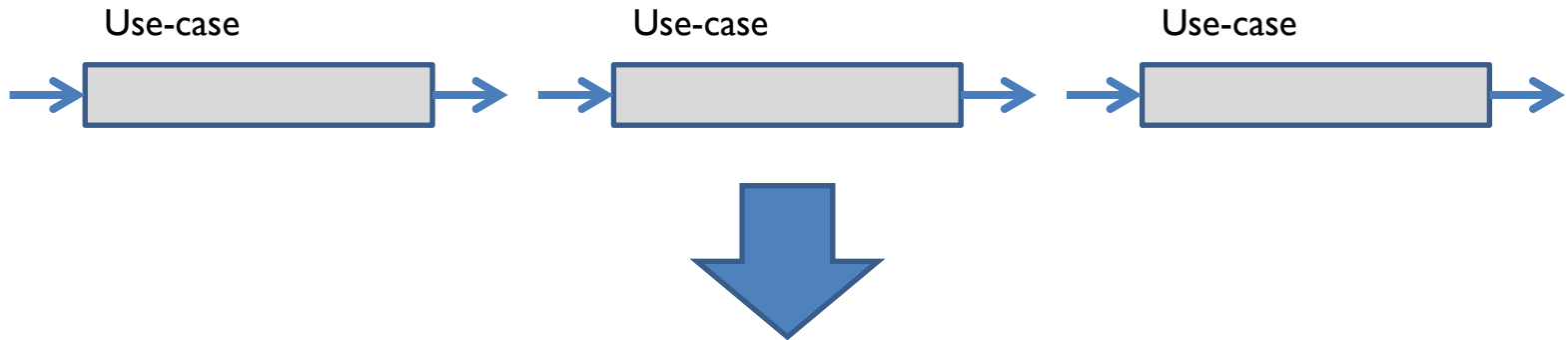


*A "Service" is just like a microservice
but without the "micro" in front*

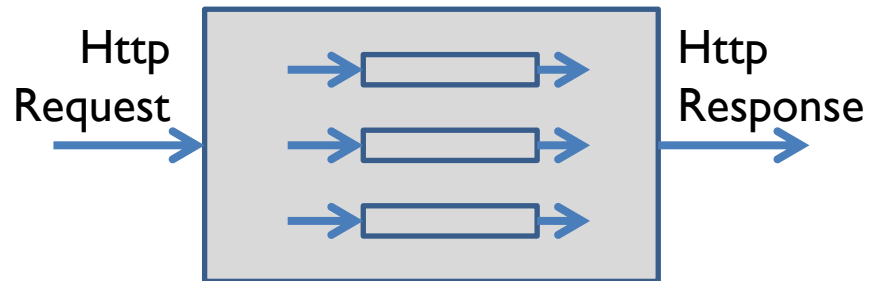


Use-case



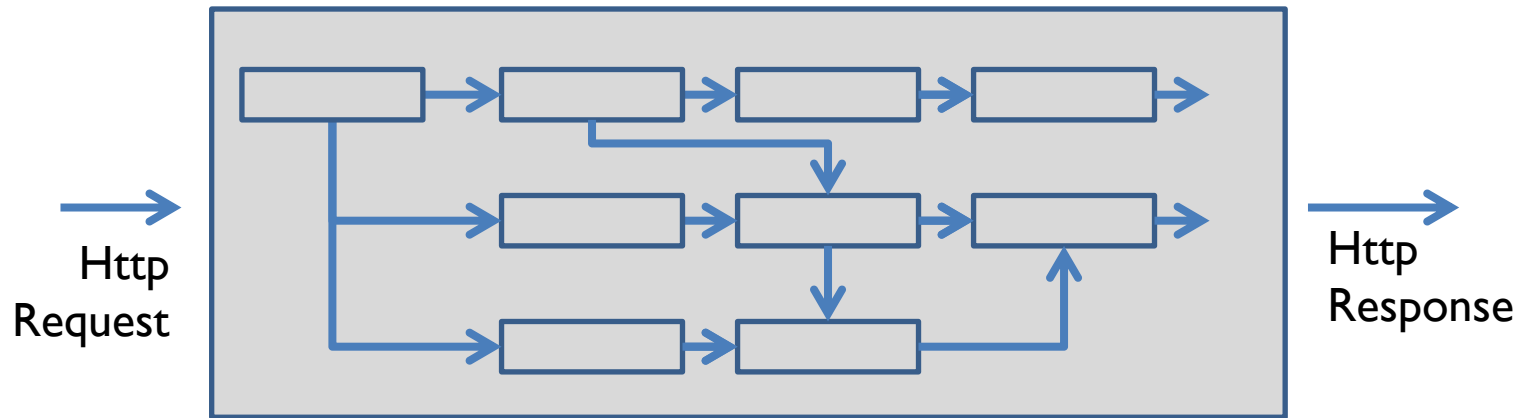


Web application

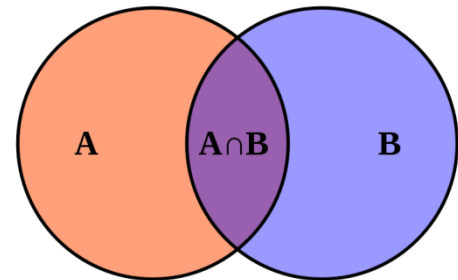


Use a dispatcher (or router,
or controller) to pick a
workflow to run

The Power of Composition

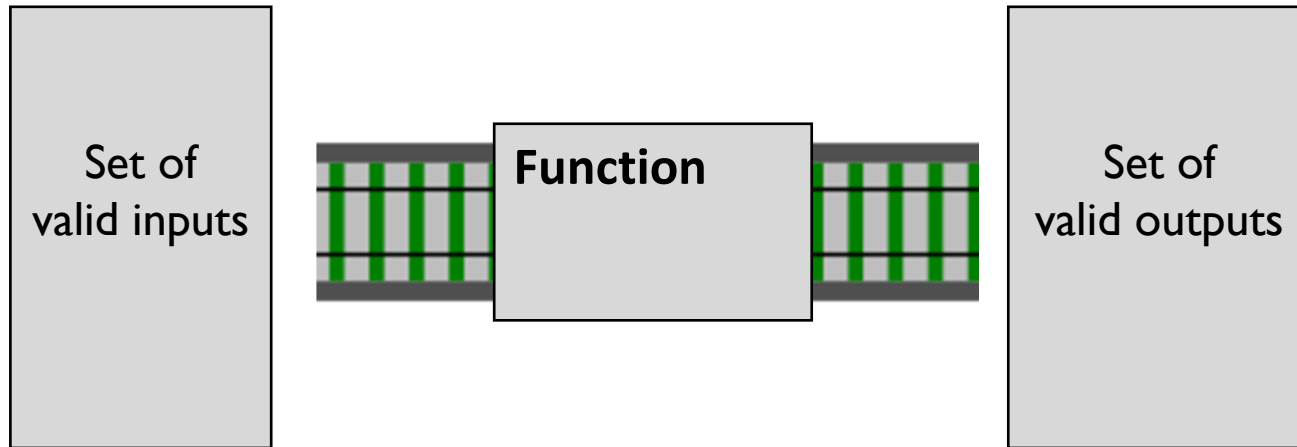


FP principle #3:
Types are not classes

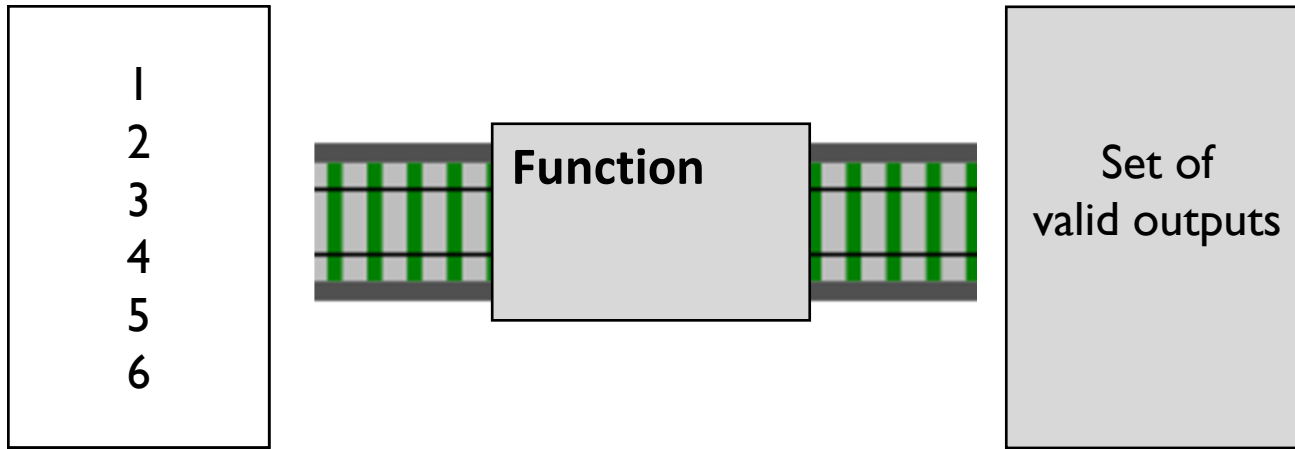


So, what is a type then?

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



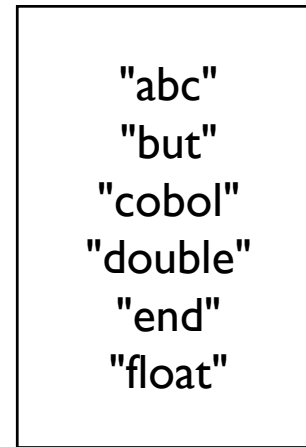
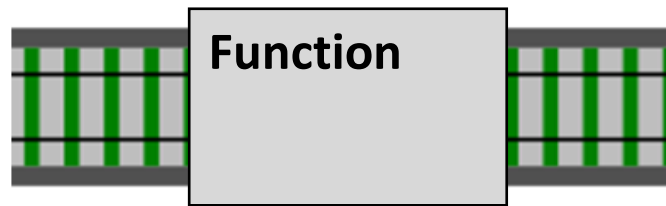
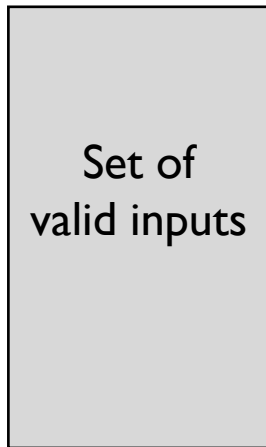
A type is a just a name
for a set of things



This is the set of all integers

type Integer

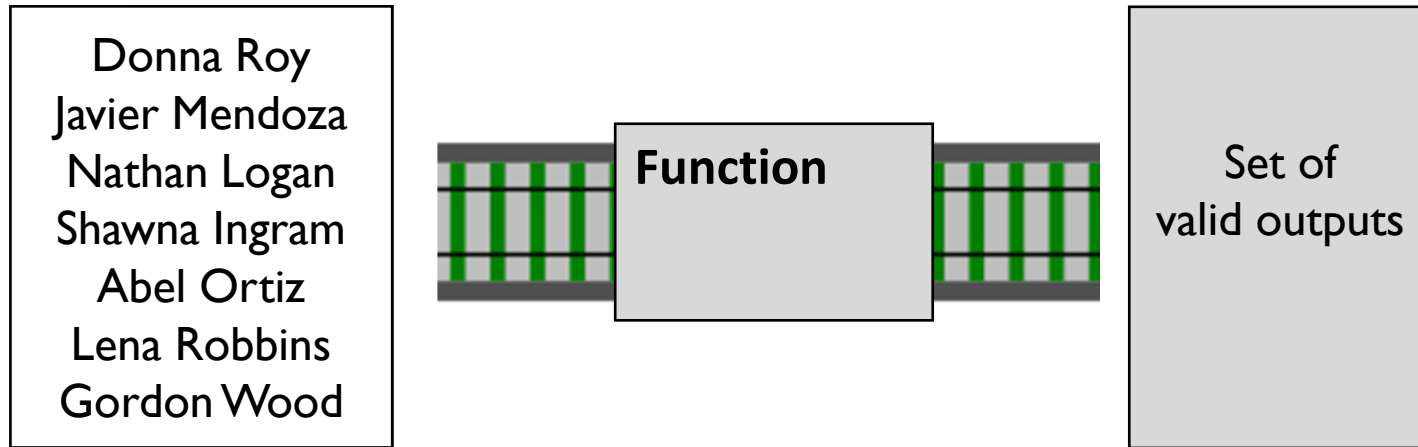
A type is a just a name
for a set of things



↑
This is the set of all strings

type String

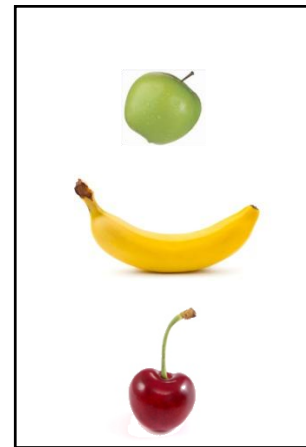
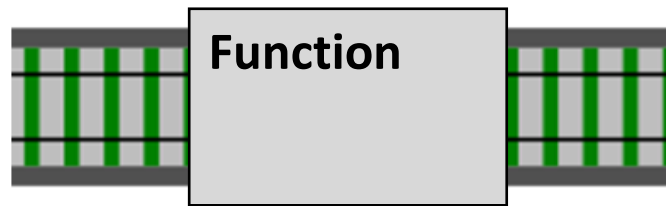
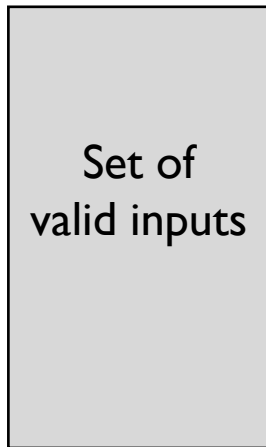
A type is a just a name
for a set of things



↑
This is the set of all people

type Person

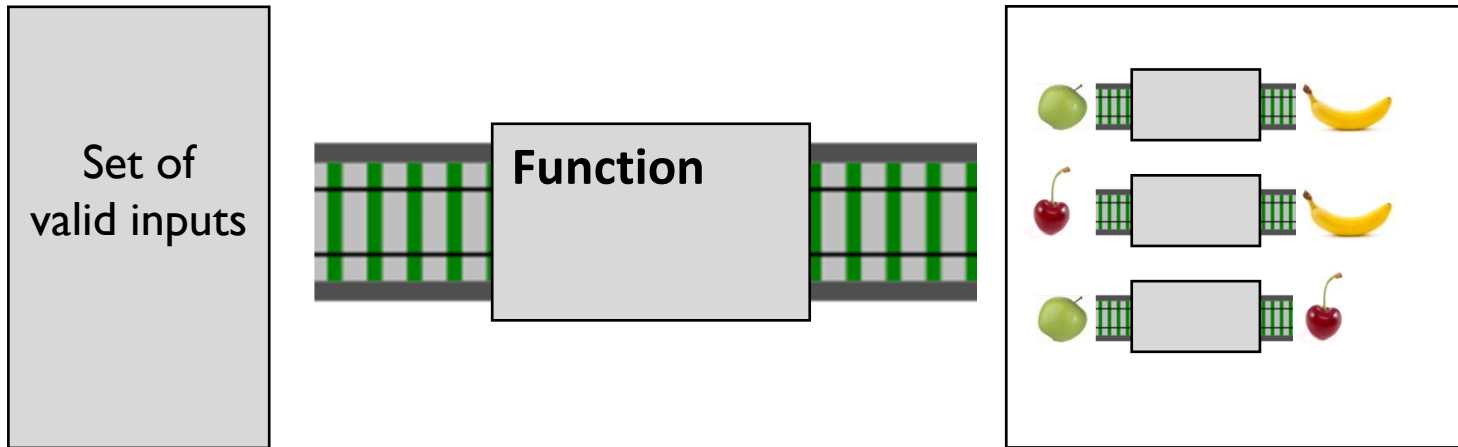
A type is a just a name
for a set of things



↑
This is the set of all fruit

type Fruit

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



↑
This is the set of all functions
with Fruit input and output

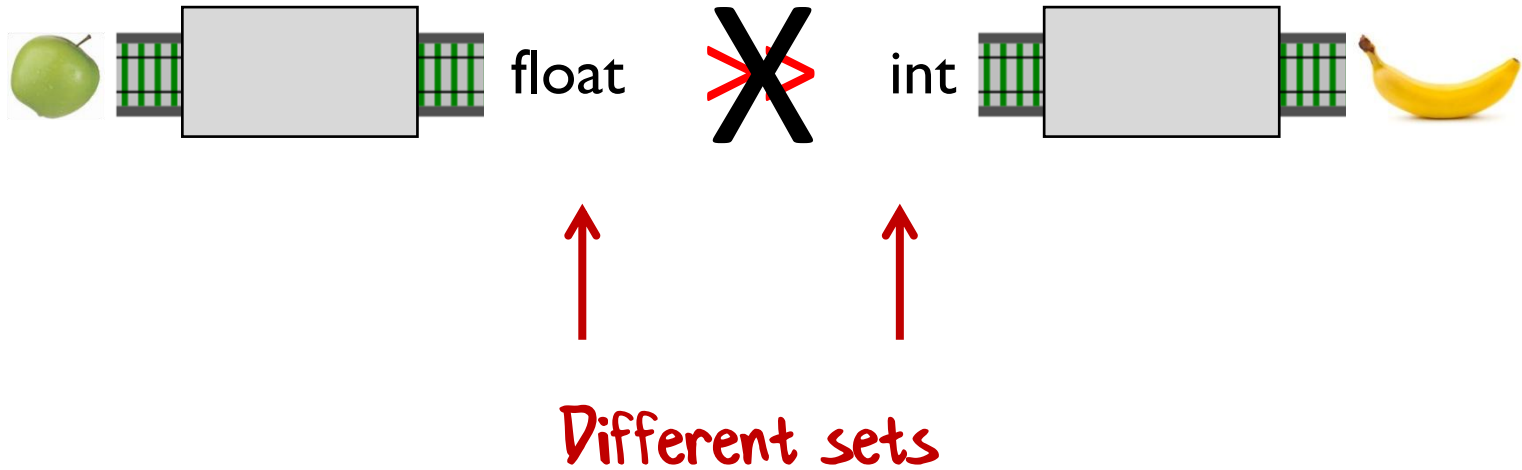
`type Fruit->Fruit`

Composition is type checked!



Different sets

Composition is type checked!



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  
}
```

*Another type, the set of
all possible apples*

←

Compose with “OR”

Snack =  or  or 

A choice type

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

Again, the set of all possible apples

A real world example of composing types

Example of some requirements:

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

For Cash we don't need any extra information

For PayPal we need an email address

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 PayPal(string emailAddress): IPaymentMethod  
{..}
```

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


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

```
type EmailAddress = string ← Primitive types
type CardNumber = string
```

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

```
type CardNumber = ...
```

Choice type
(using OR)



```
type CardType = Visa | Mastercard
```

```
type CreditCardInfo = {
```

```
  CardType : CardType
```

```
  CardNumber : CardNumber
```

```
}
```

Record type (using AND)



```
type EmailAddress = ...  
type CardNumber = ...  
type CardType = ...  
type CreditCardInfo = ...
```

```
type PaymentMethod =
```

```
| Cash
```

```
| PayPal of EmailAddress
```

```
| Card of CreditCardInfo
```

 *Choice type*

```
type EmailAddress = ...
type CardNumber = ...
type CardType = ...
type CreditCardInfo = ...
type PaymentMethod =
  | Cash
  | PayPal of EmailAddress
  | Card of CreditCardInfo
```

```
type PaymentAmount = decimal
```

```
type Currency = EUR | USD
```

Another primitive type

Another choice type

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

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

 **Record type**

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

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

Final type built from many
smaller types:

The Power of Composition



The End

This is everything you need to know
about functional programming!

Part II

Composition in practice

Composition

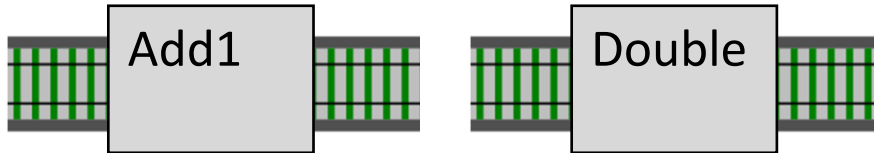
```
let add1 x = x + 1
let double x = x + x
let square x = x * x
```

```
// in most programming languages,
// composition looks like this...
```

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

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

Composition



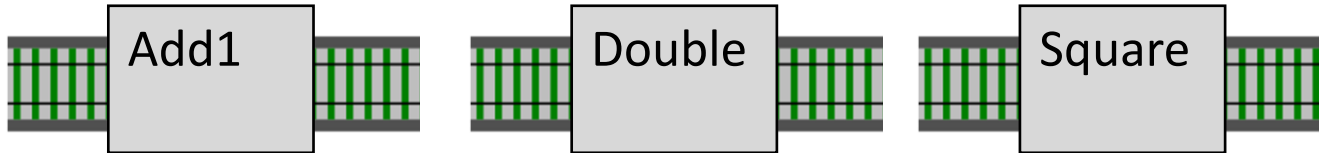
```
let add1 x = x + 1
let double x = x + x
```

```
// in F#, composition looks like this
let add1_double = add1 >> double
```

Compositon
operator

```
// call it
add1_double 5 // 12
```

Composition

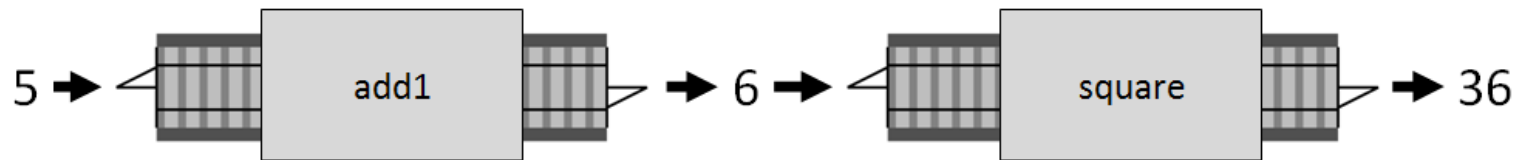


```
let add1_double_square =  
    add1 >> double >> square
```

```
// call it
```

```
add1_double_square 5    // 144
```

Piping





Pipe symbol

This is easier to understand

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

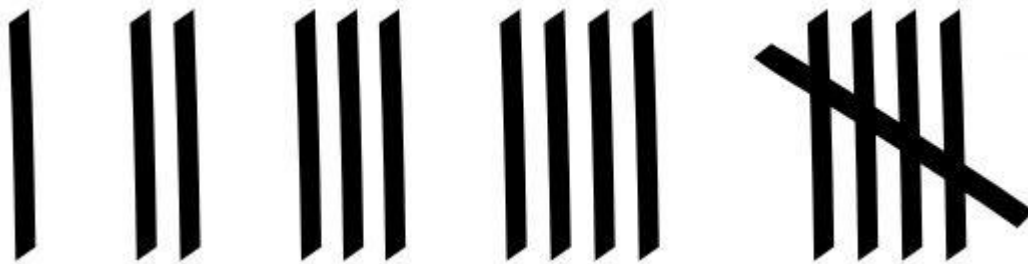
pipe

Demo #1: Piping

Composition in practice:
Roman Numerals Converter

To Roman Numerals

- Task: convert an integer to Roman Numerals
- 5 => "V" 10 => "X" 100 => "C" etc

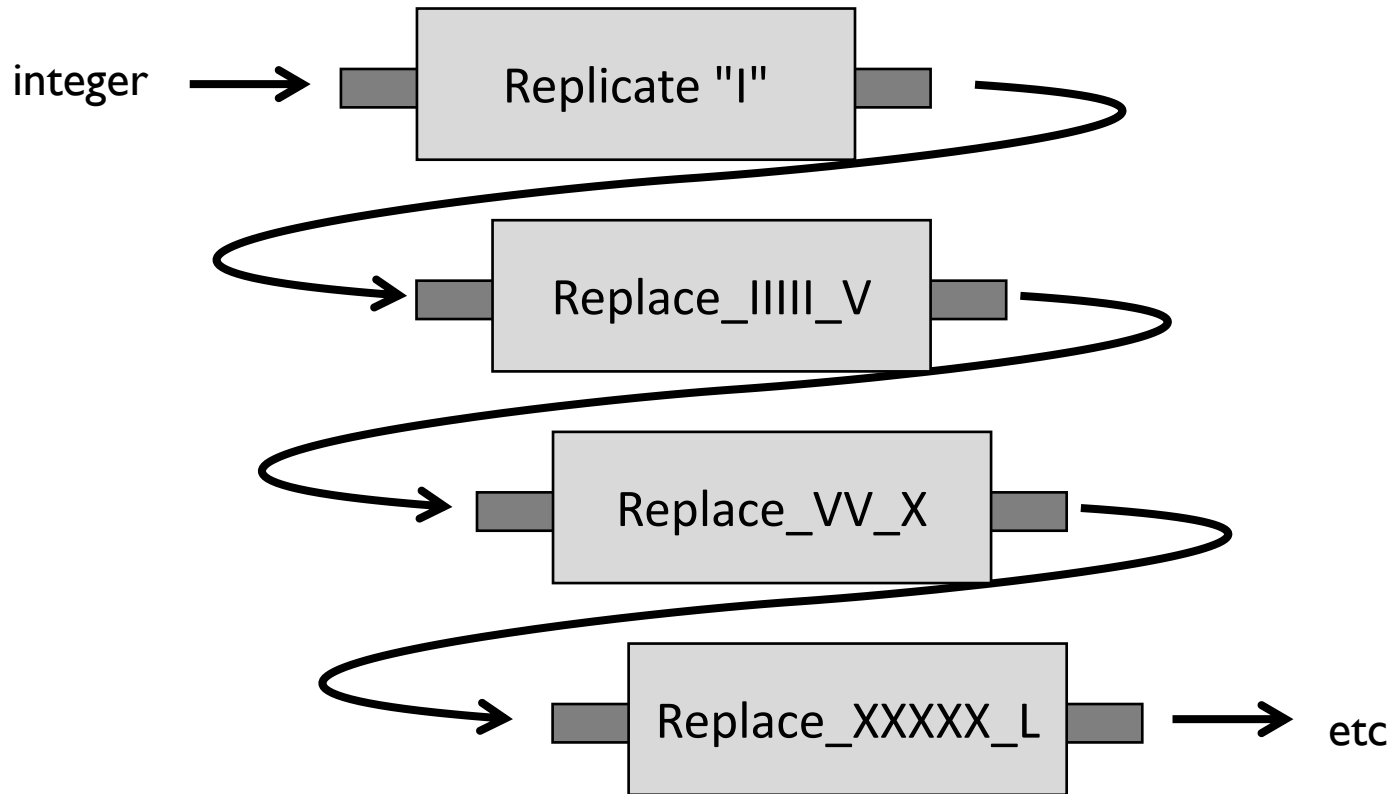


*Roman numbers evolved
from this*

To Roman Numerals

- Use the "tally" approach
 - Start with N copies of "I"
 - Replace five "I"s with a "V"
 - Replace two "V"s with a "X"
 - Replace five "X"s with a "L"
 - Replace two "L"s with a "C"
 - Replace five "C"s with a "D"
 - Replace two "D"s with a "M"

Implementation using piping



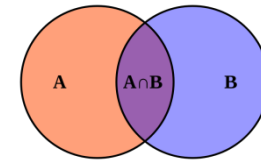
Demo #2: Roman Numerals

We covered all the core principles of FP

Functions



Types



Composition everywhere



But what if something goes wrong 😞

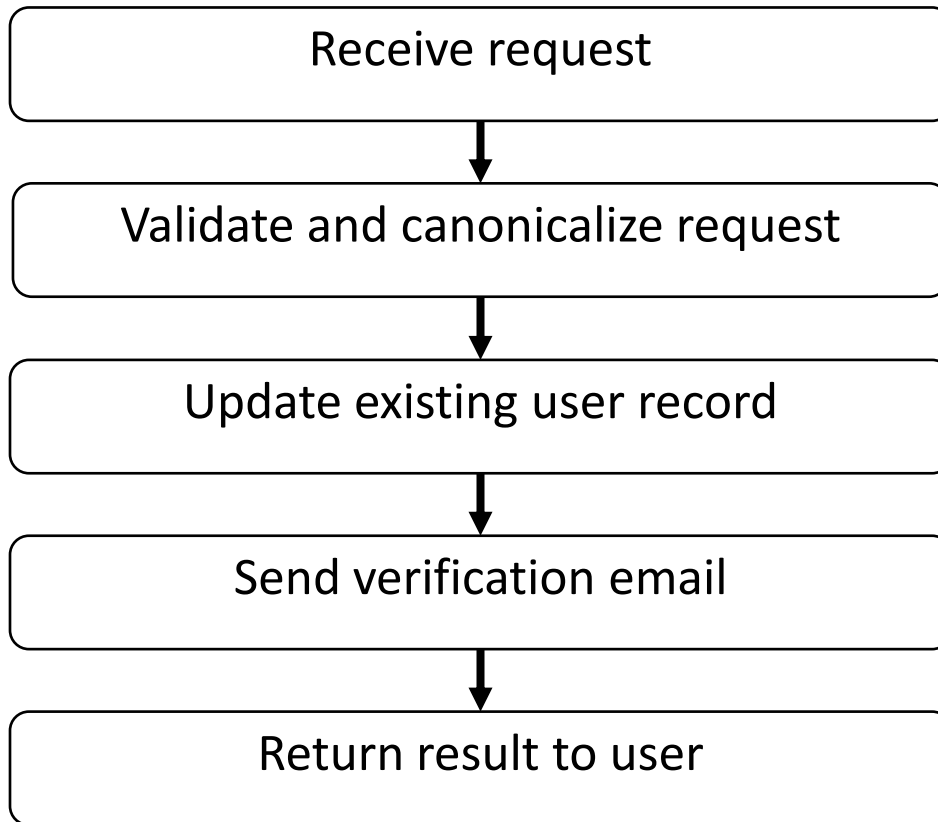
Part III

Error handling in functional programming

We like to focus on
happy path programming



"As a user I want to update my name and email address"




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

Straying from the happy path...

What do you do when
something goes wrong?

Microsoft Visual Studio

 An exception of type 'System.NotImplementedException' occurred in UnhandledExceptionBlog.exe but was not handled in user code

Additional information: The developer needs to do his job.


Form1

 Unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will close immediately.

ORA-1017: invalid username/password; logon denied.


Details Continue Quit

Error

 An error has occurred while creating an error report

OK

Error

 The operation completed successfully.

OK

Microsoft Money

 An error has occurred but the error message cannot be retrieved due to another error.

OK

Error

occurred in your
ne, the
to contin
shut dow

Details Continue

==== Exception Text =====
n.IO.IOException: The device is not ready.

ystem.IO._Error.WinIOError(Int
ystem.IO.FileStream..ctor(String
ystem.IO.FileStream..ctor(String
rorHandling.frmErrors.NoErrorH
at ErrorHandler.frmErrors.btnErrorH
at System.Windows.Forms.Control.C
at System.Windows.Forms.Button.O

Keyboard not plugged

 Windows 95 was unable to detect your keyboard. Press F1 to retry or F2 to abort.

Microsoft Visual Basic

Run-time error '6':
Overflow

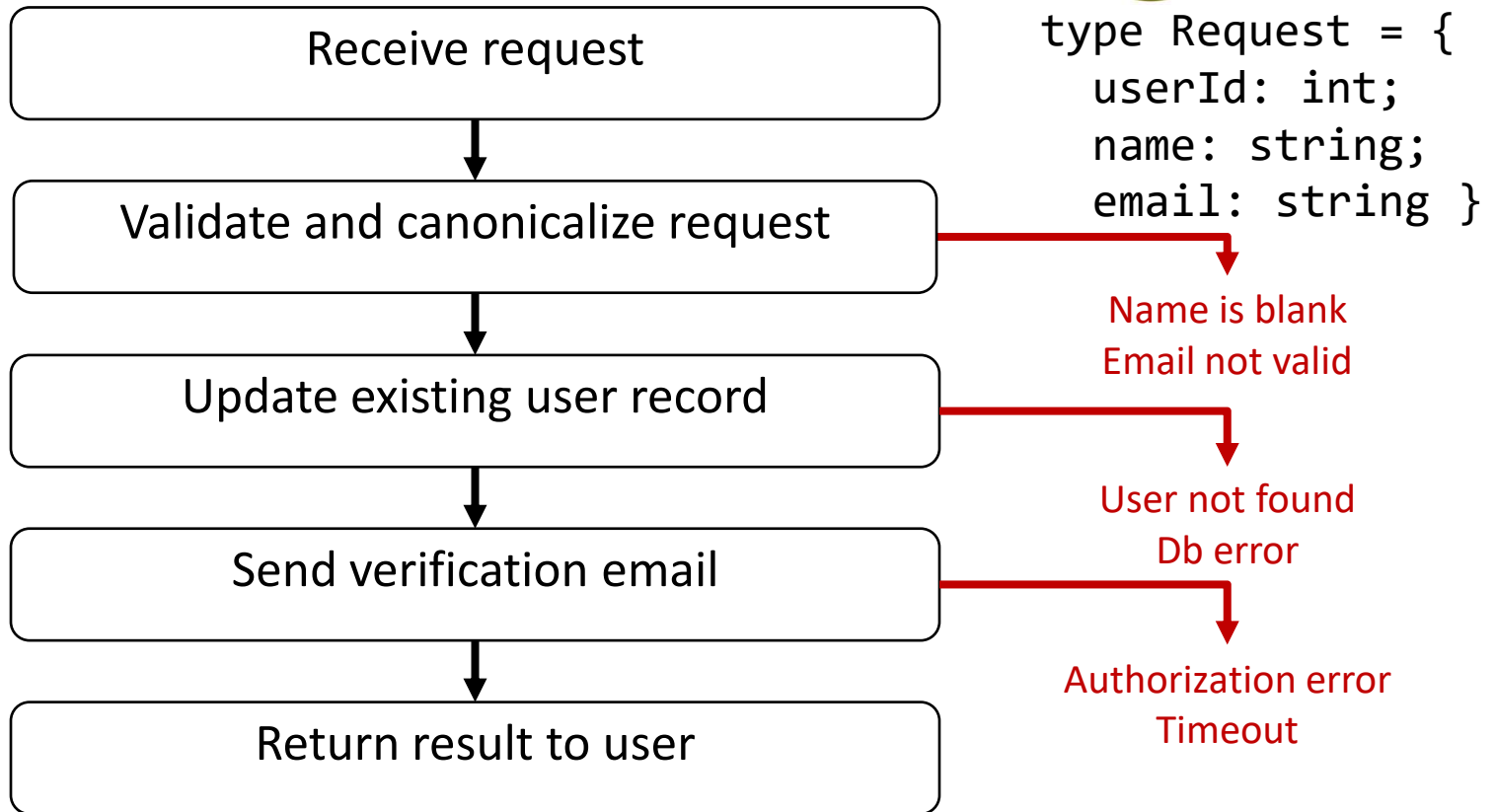
Continue End Debug Help

NO! - Bad User!!!

 You've been warned 3 times that this file does not exist. Now you've made us catch this worthless exception and we're upset. Do not do this again.

OK

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



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

    return "OK";
}
```

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

    return "OK";
}
```

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

    smtpServer.sendEmail(request.Email)

    return "OK";
}
```



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

    smtpServer.sendEmail(request.Email)

    return "OK";
}
```

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

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

    return "OK";
}
```

```

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

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

    return "OK";
}

```

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

Q: What is the functional equivalent of this code?

Can we preserve the elegance of the original functional version?

Using a *Result* type for error handling



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

Define a choice type details of the error

But this is not generic, so...



```
type Result<'SuccessType, 'ErrorType> =  
    | Ok of 'SuccessType  
    | Error of 'ErrorType
```

A generic, reusable Result type

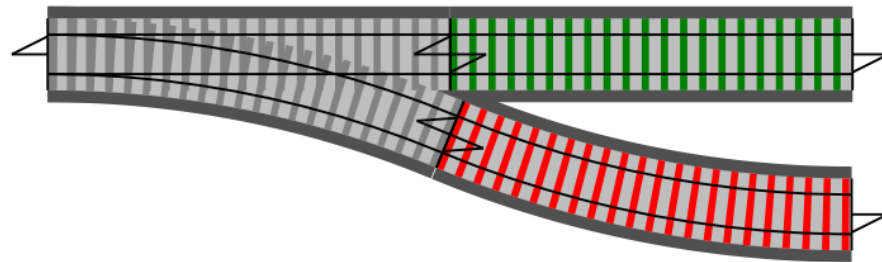
This type is built-in to F#, but if not, you could always define it yourself.



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

*We have data for both
success and failure paths*

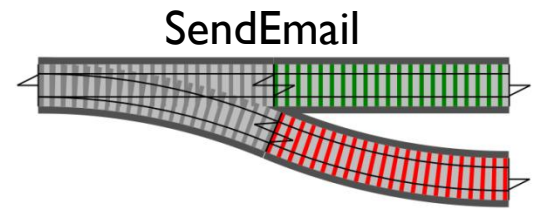
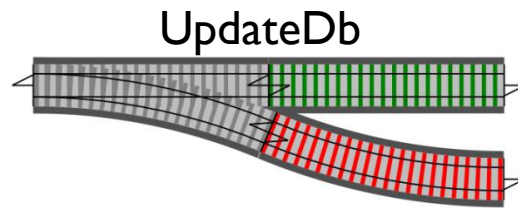
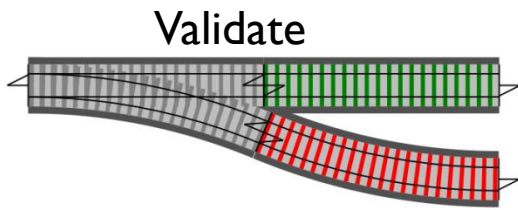
Input ->



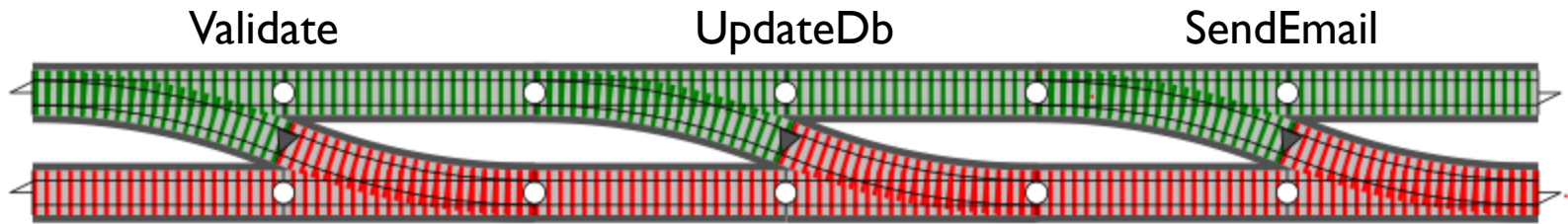
Success!

Failure

We design all functions to return choices like this...



...and then compose them together into a pipeline



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

Functional flow without error handling

Before error handling

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

One track

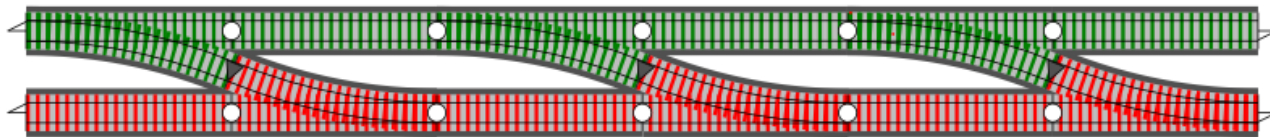


Functional flow with error handling

After

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

Two track

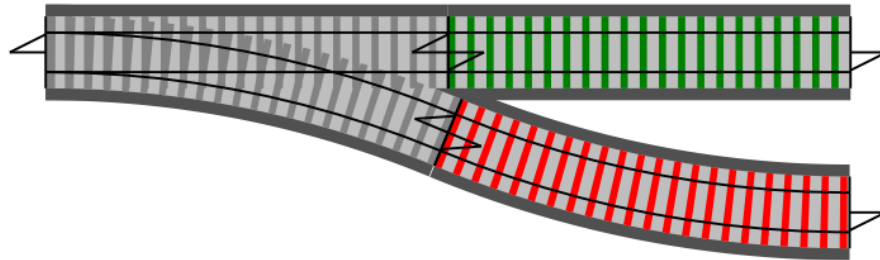


I will show you this code soon!

How to compose railway functions

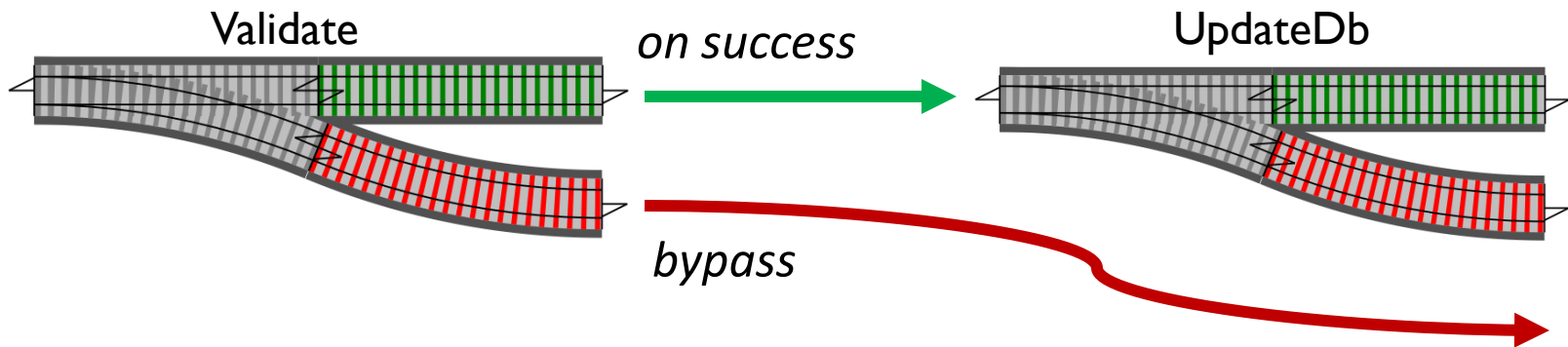
a.k.a. "Railway Oriented Programming"

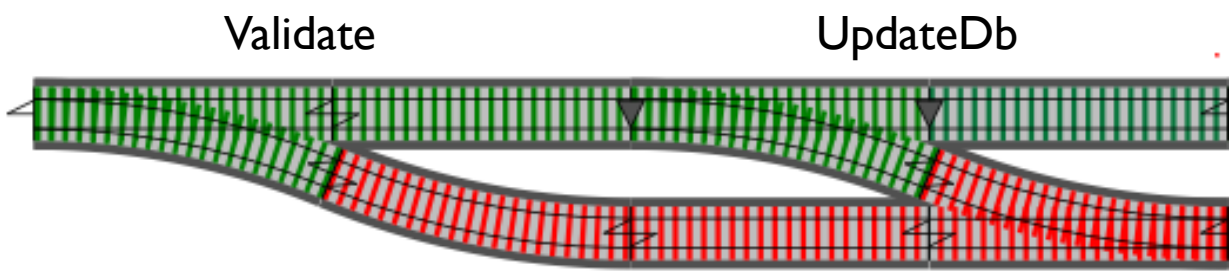
Input ->

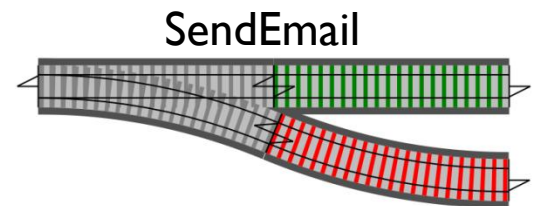
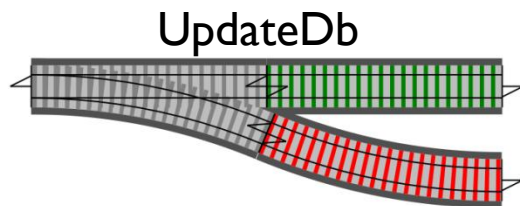
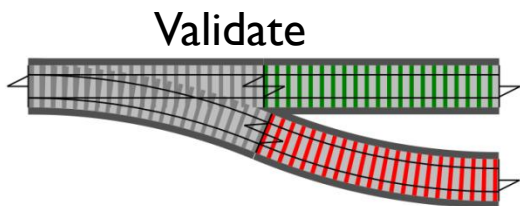


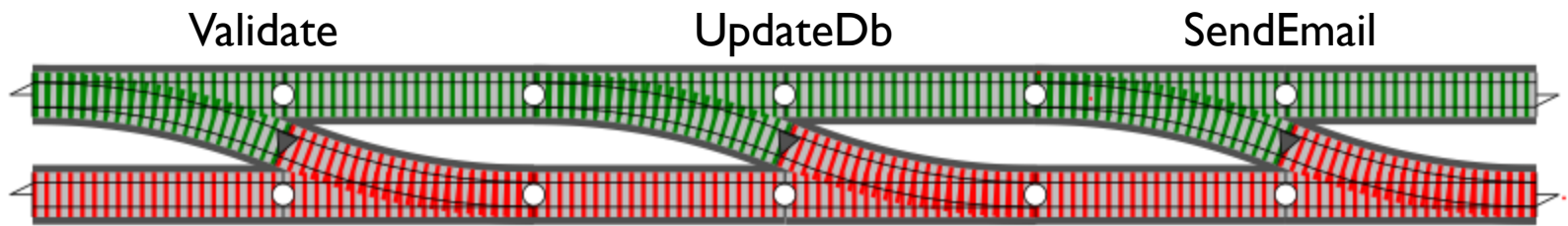
Success!

Failure





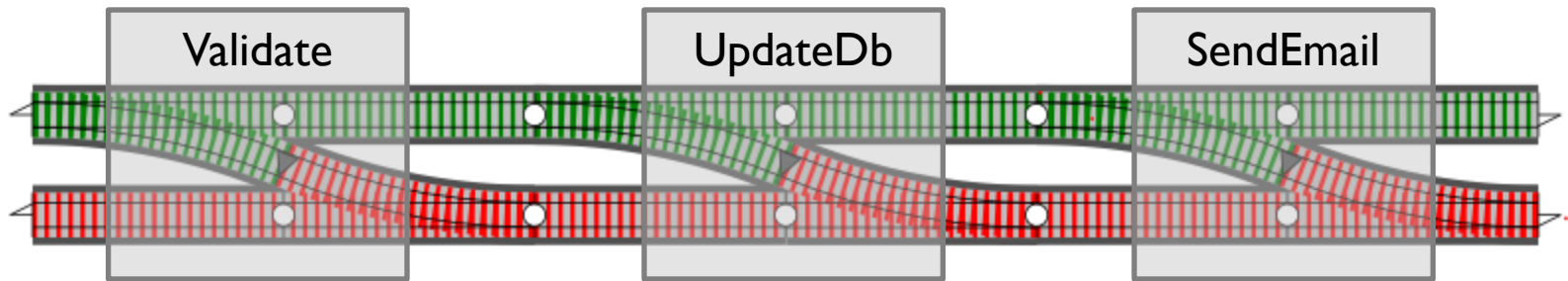




Composition is obvious in theory!

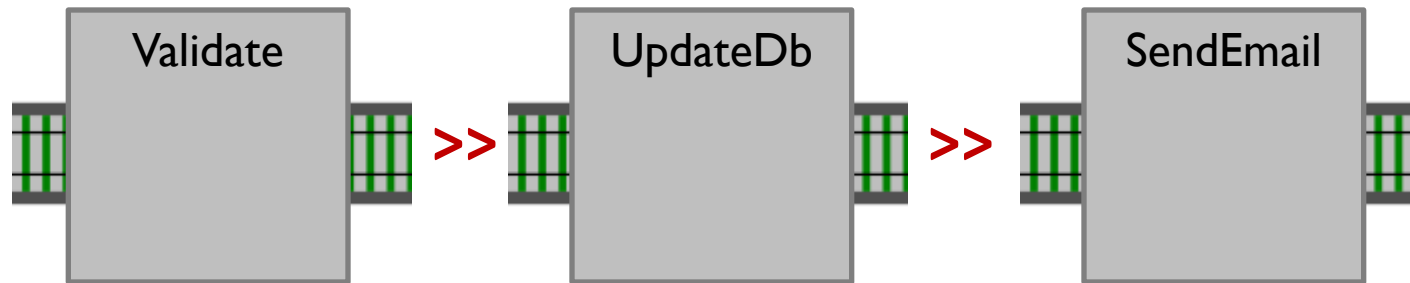
But how do you do it in practice?

**How to compose
these functions?**

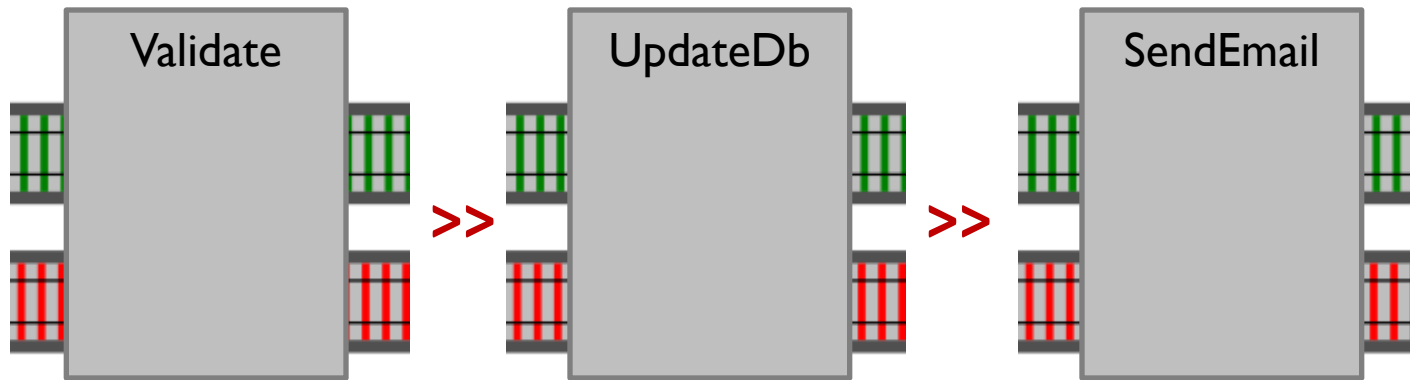


Here we have a series of black box functions that are straddling a two-track railway.

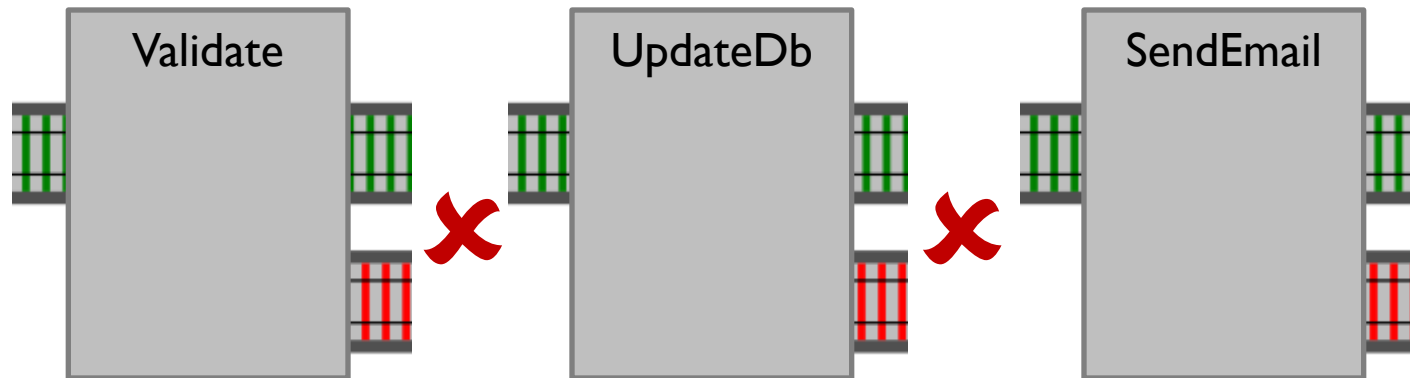
Inside each box there is a switch function.



Composing one-track functions is fine...

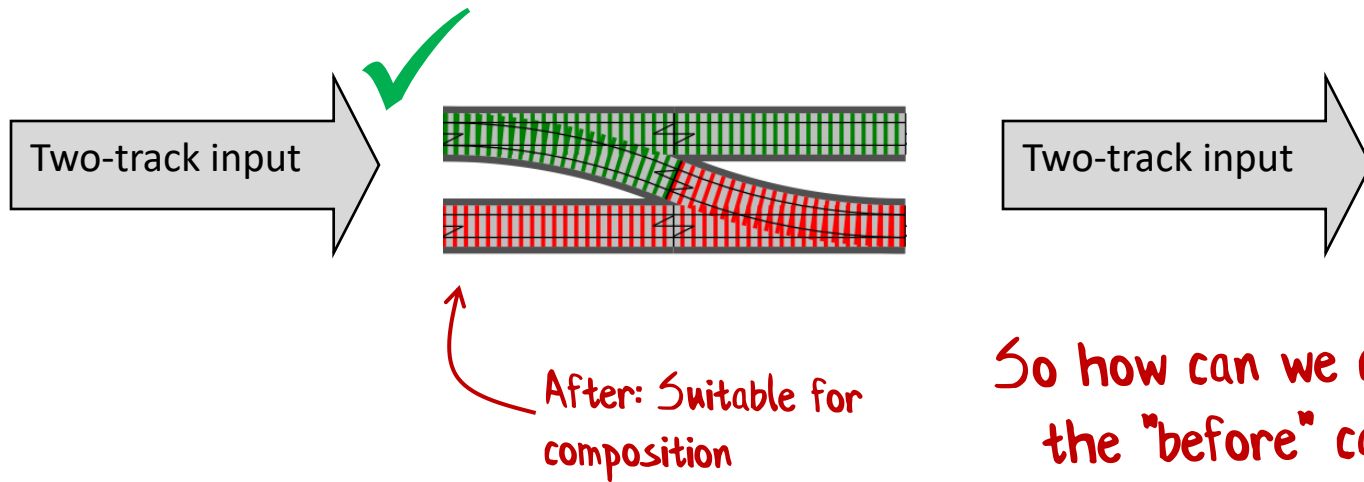
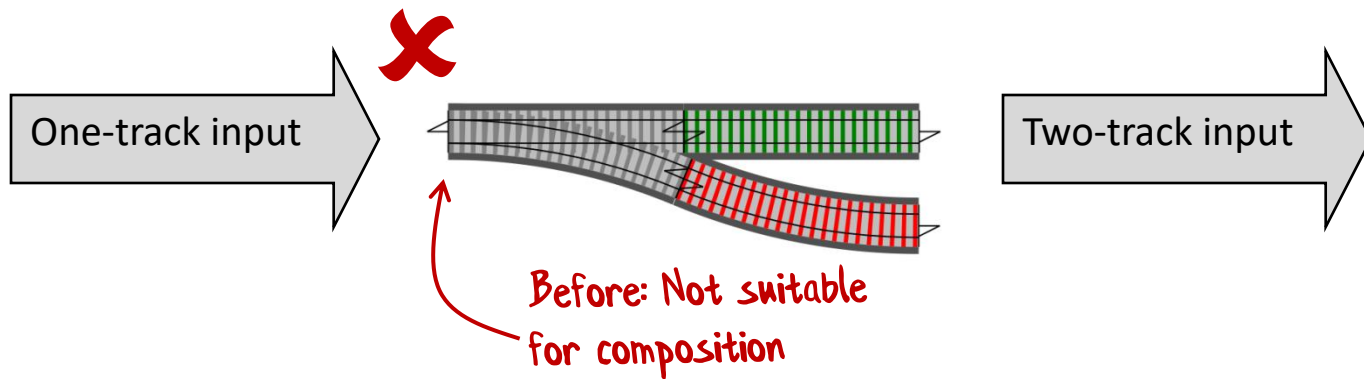


... and composing two-track functions is fine...

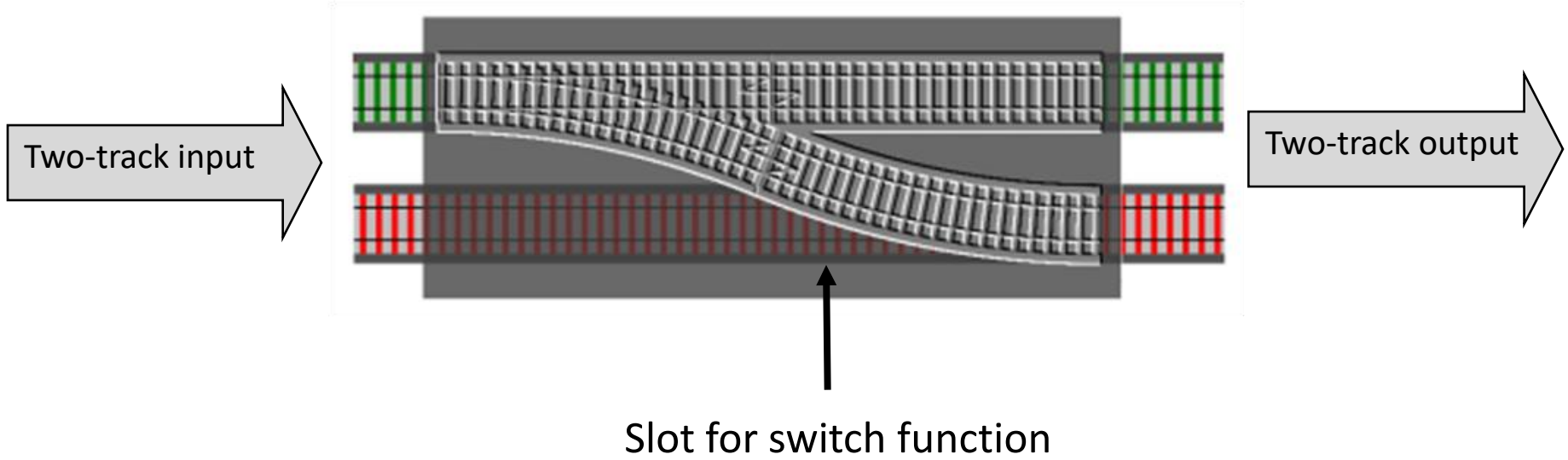


... but composing switches is not allowed!

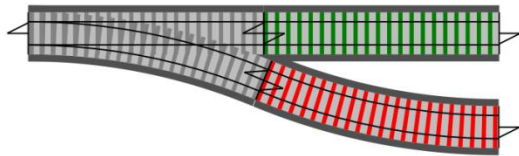
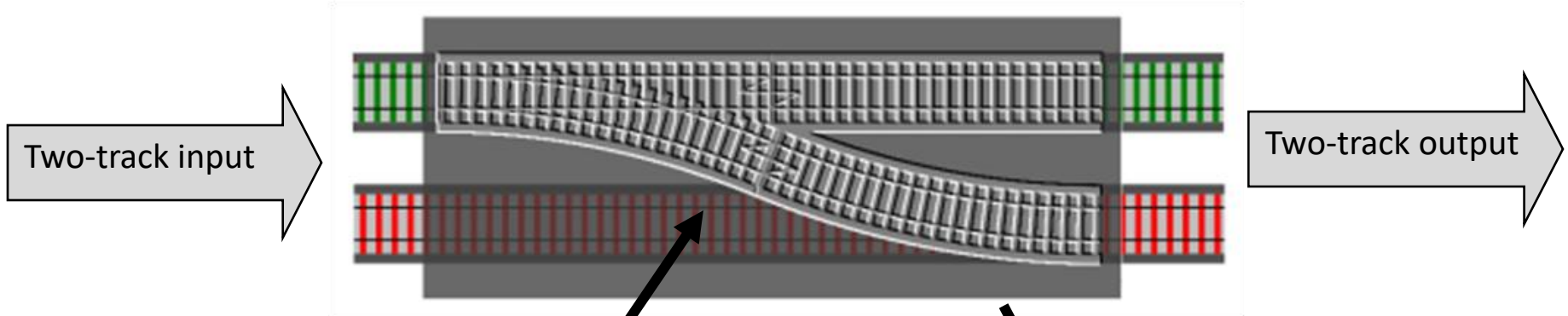
**How to combine the
mismatched functions?**



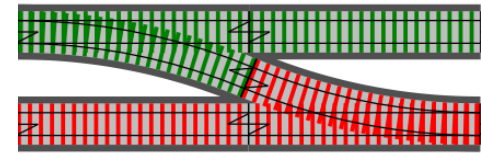
So how can we convert from the "before" case to the "after" case?



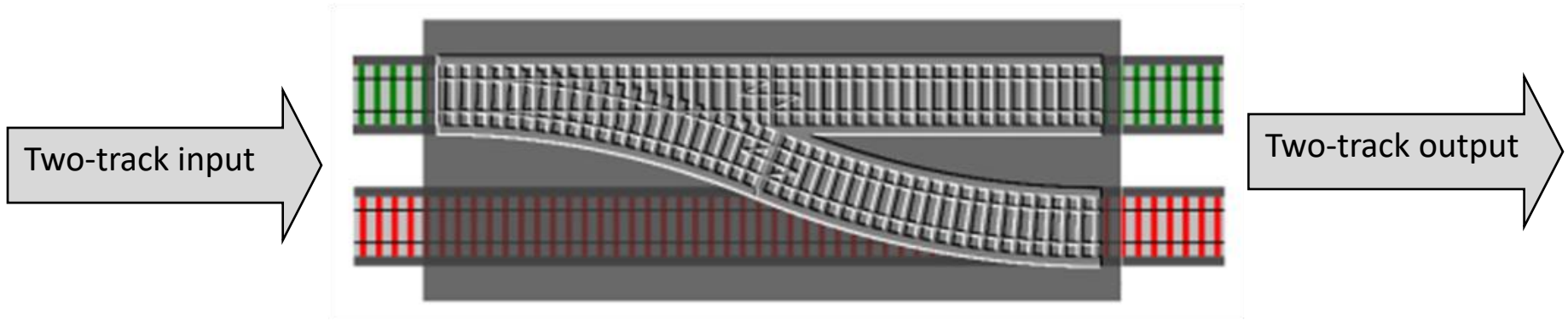
This adapter block is
a "function transformer"



A function as input

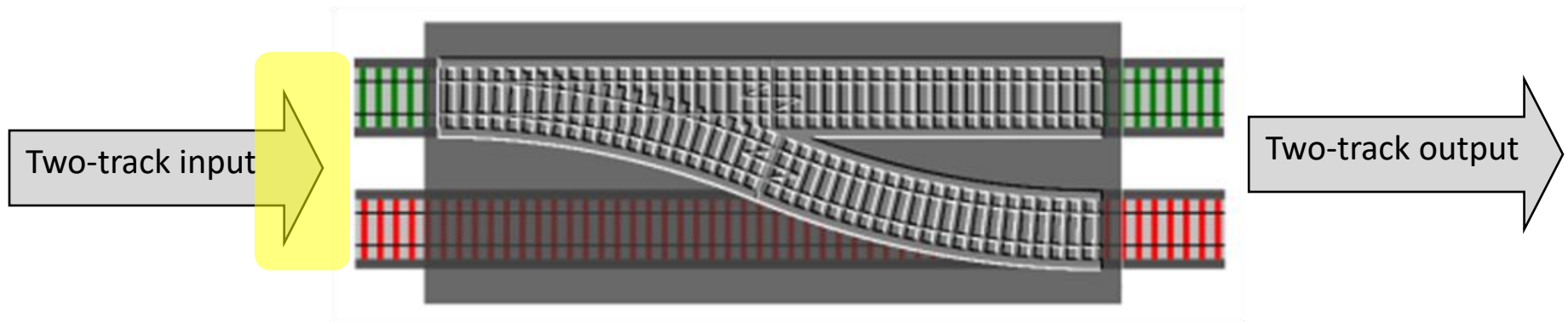


A different shape function as output



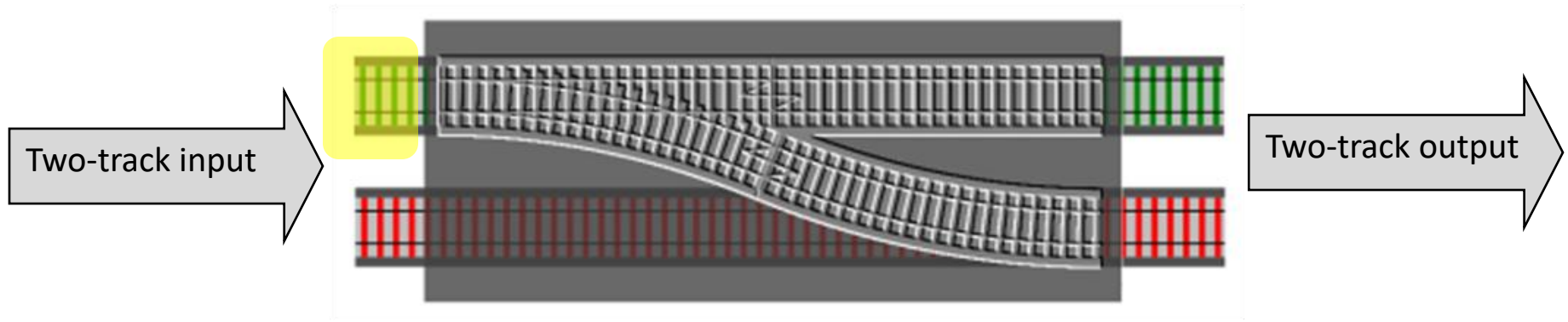
```
let bind nextFunction result =  
  match result with  
  | Ok s -> nextFunction s  
  | Error e -> Error e
```

*"bind" is the common name for this kind of function.
Also called "flatMap" or "andThen" or ">>=".
In LINQ it's "SelectMany"*



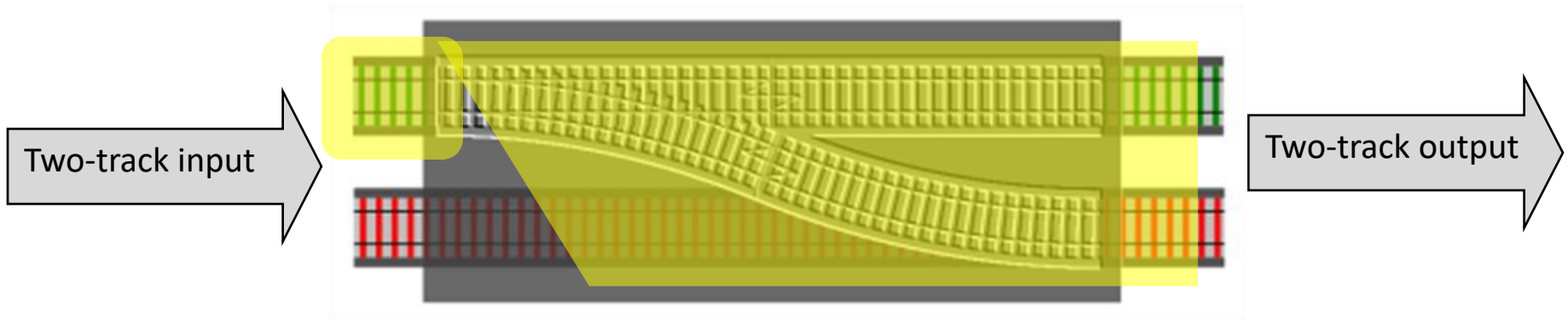
```
let bind nextFunction result =  
  match result with  
  | Ok s -> nextFunction s  
  | Error e -> Error e
```

The input is a two-track value

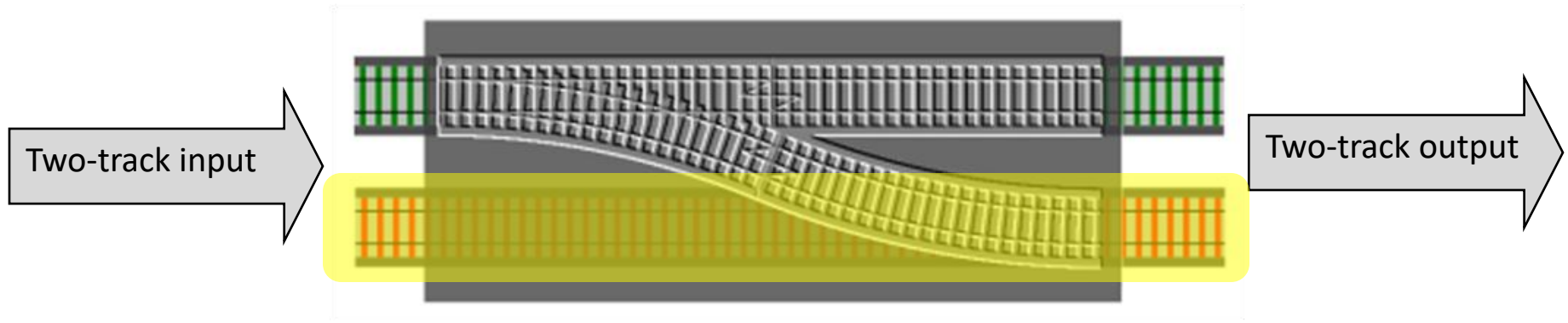


```
let bind nextFunction result =  
  match result with  
  | Ok s -> nextFunction s  
  | Error e -> Error e
```

a) handle the Ok track



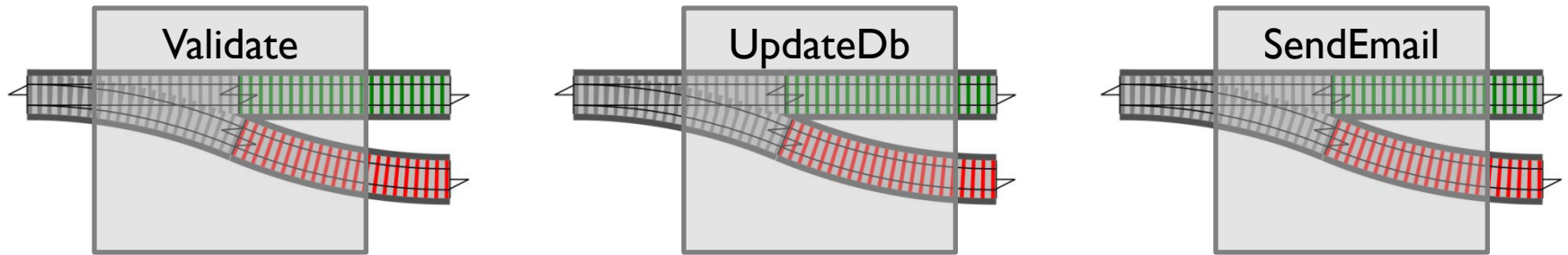
```
let bind nextFunction result =  
  match result with  
  | Ok s -> nextFunction s  
  | Error e -> Error e
```



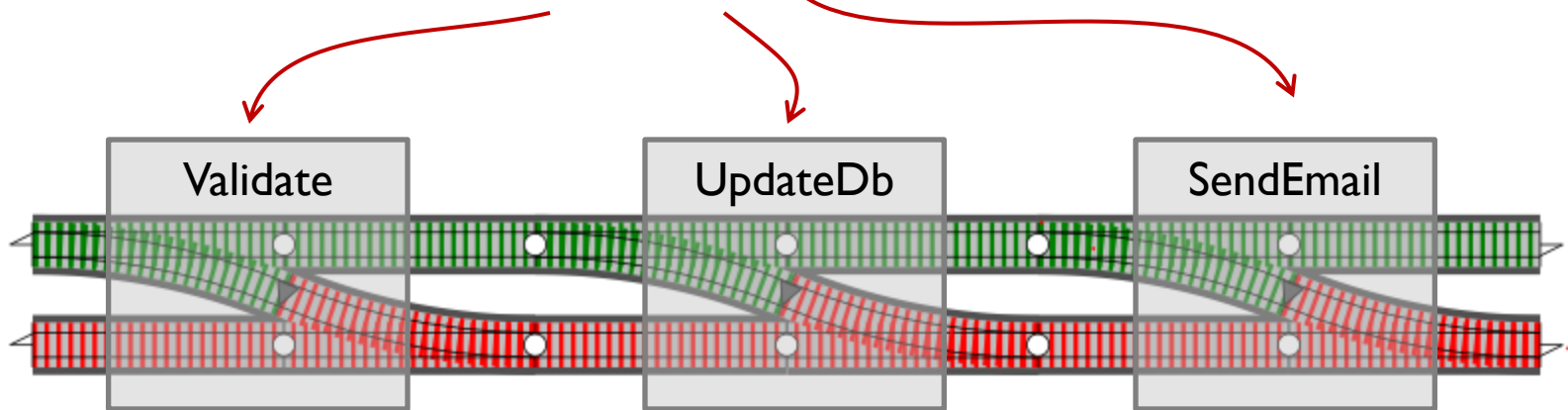
```
let bind nextFunction result =  
  match result with  
  | Ok s -> nextFunction s  
  | Error e -> Error e
```

b) Handle the Error track

Composing switches - review



Converted to two-track functions using bind



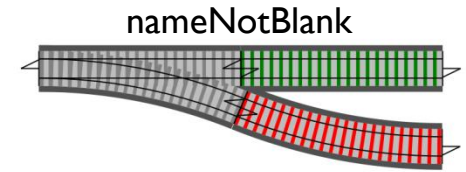
Bind example

Validating input

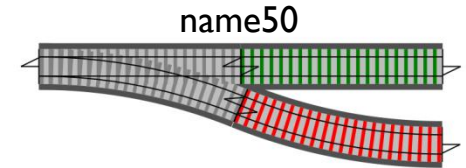
```
type Request = {  
  Name : string  
  Email : string  
}
```

Is this data valid?

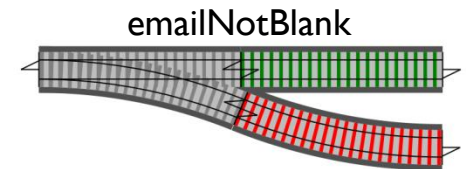
```
let checkNameNotBlank input =  
  if input.Name = "" then  
    Error "Name must not be blank"  
  else  
    Ok input
```

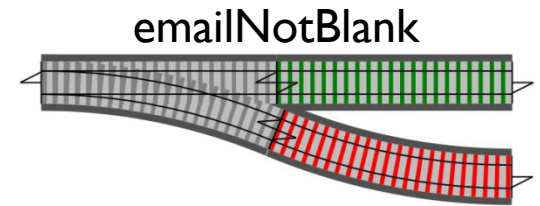
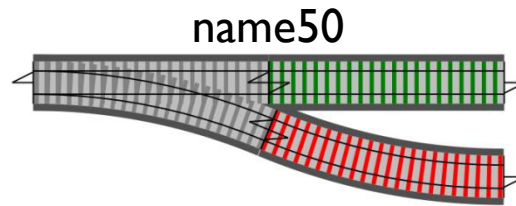
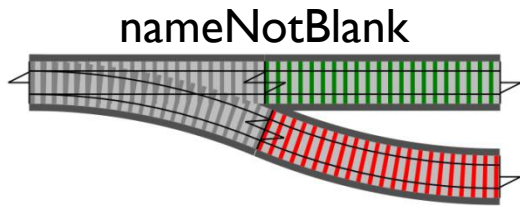


```
let checkName50 input =  
  if input.Name.Length > 50 then  
    Error "Name must not be longer than 50 chars"  
  else  
    Ok input
```

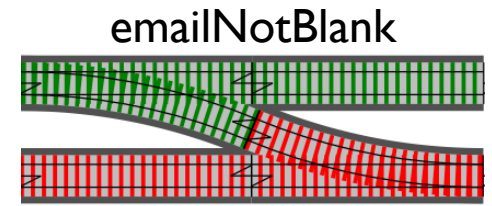
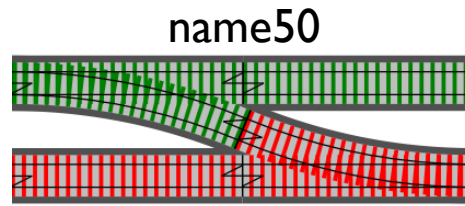
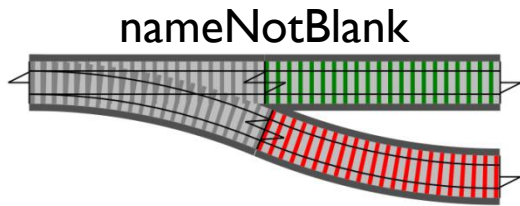


```
let checkEmailNotBlank input =  
  if input.Email = "" then  
    Error "Email must not be blank"  
  else  
    Ok input
```





checkNameNotBlank (composed with)
 checkName50 (composed with)
 checkEmailNotBlank

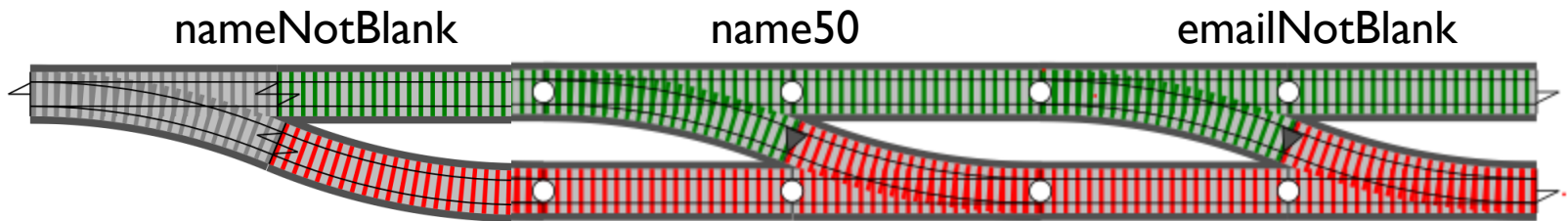


checkNameNotBlank

bind checkName50

bind checkEmailNotBlank

use "bind" to
convert to 2-track



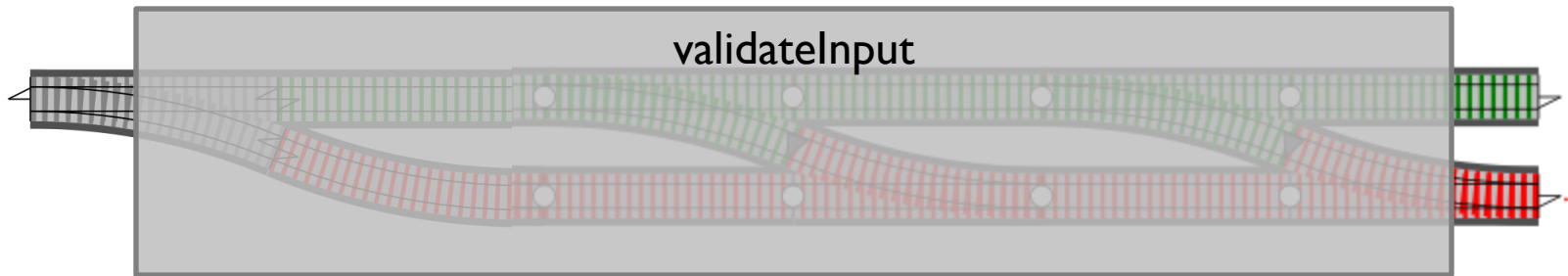
request

```
|> checkNameNotBlank
```

```
|> Result.bind checkName50
```

```
|> Result.bind checkEmailNotBlank
```

then compose together

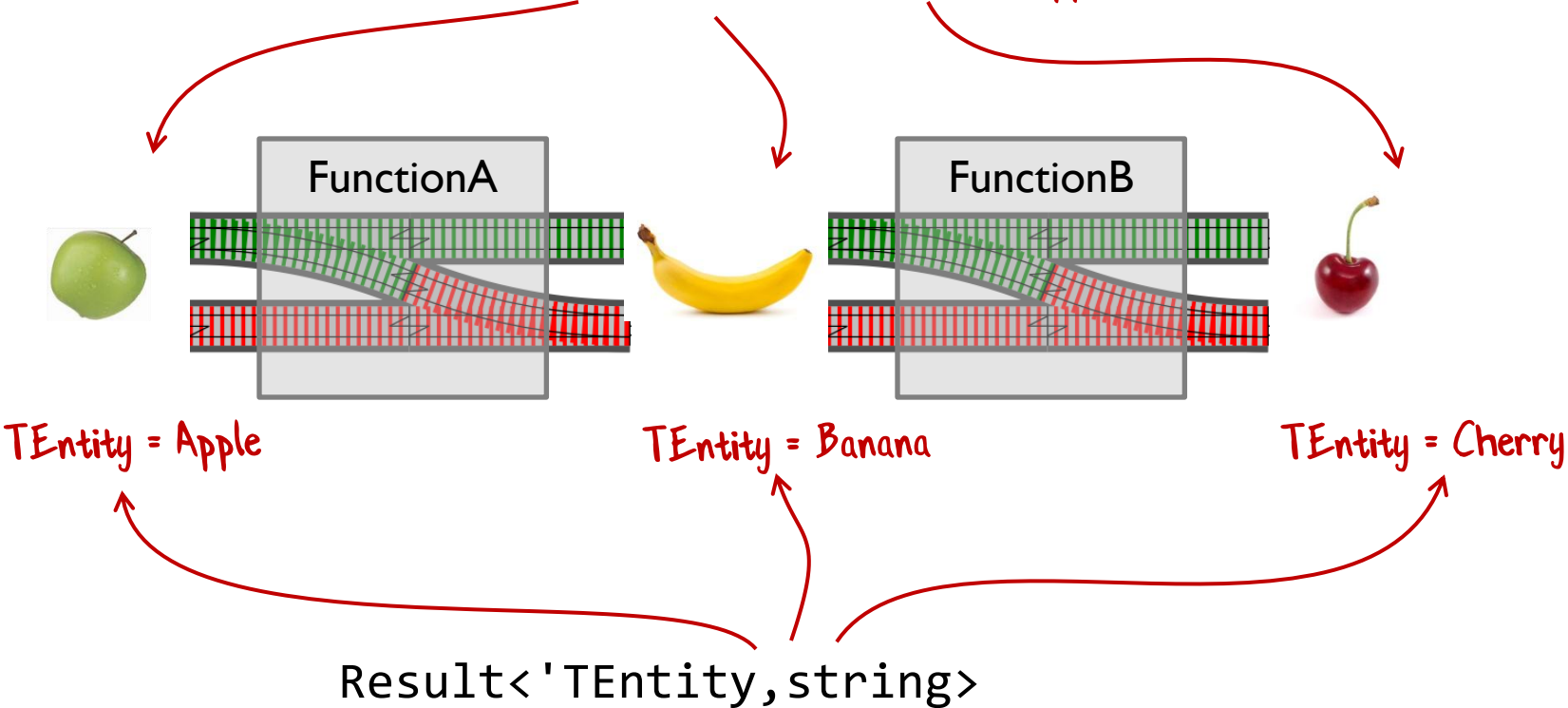


Define a function

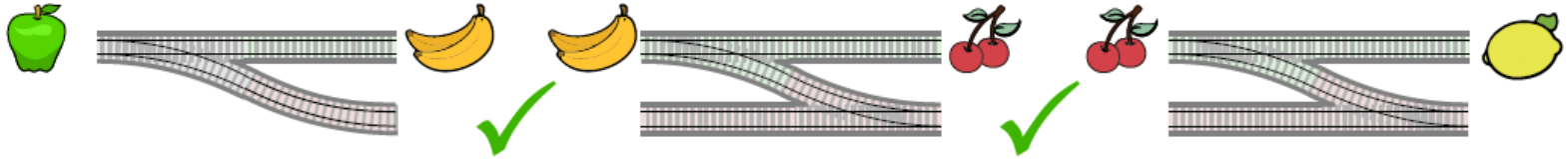
```
let validateInput input =  
  input  
  |> checkNameNotBlank  
  |> Result.bind checkName50  
  |> Result.bind checkEmailNotBlank
```

*Overall result is a new
two-track function*

Note that bind is about shape.
Transformations can still happen.



Type checking still applies



Demo #3:

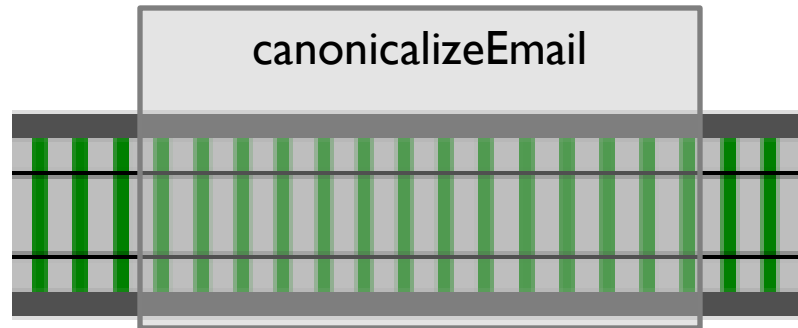
Bind example

Part IV:

Handling different "shapes" of track

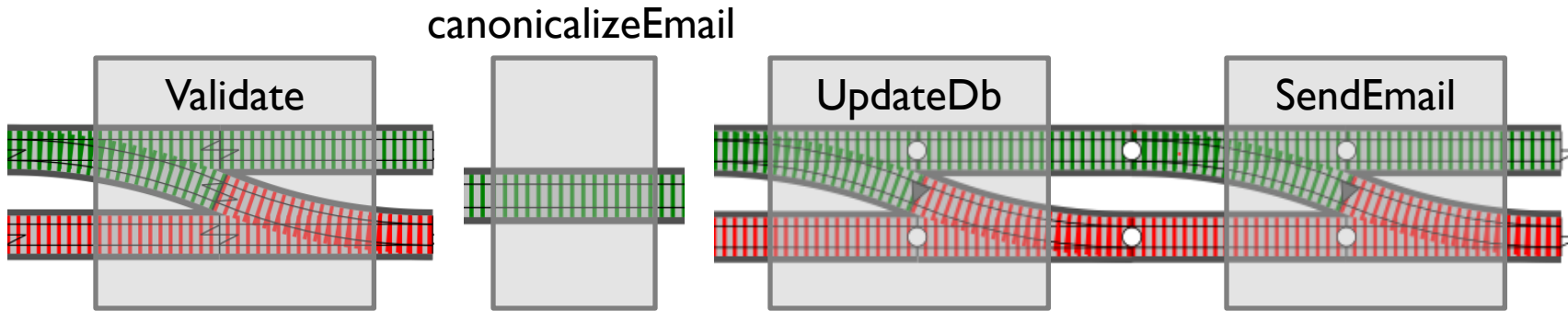
Not everything looks like that!

Working with "one-track" functions

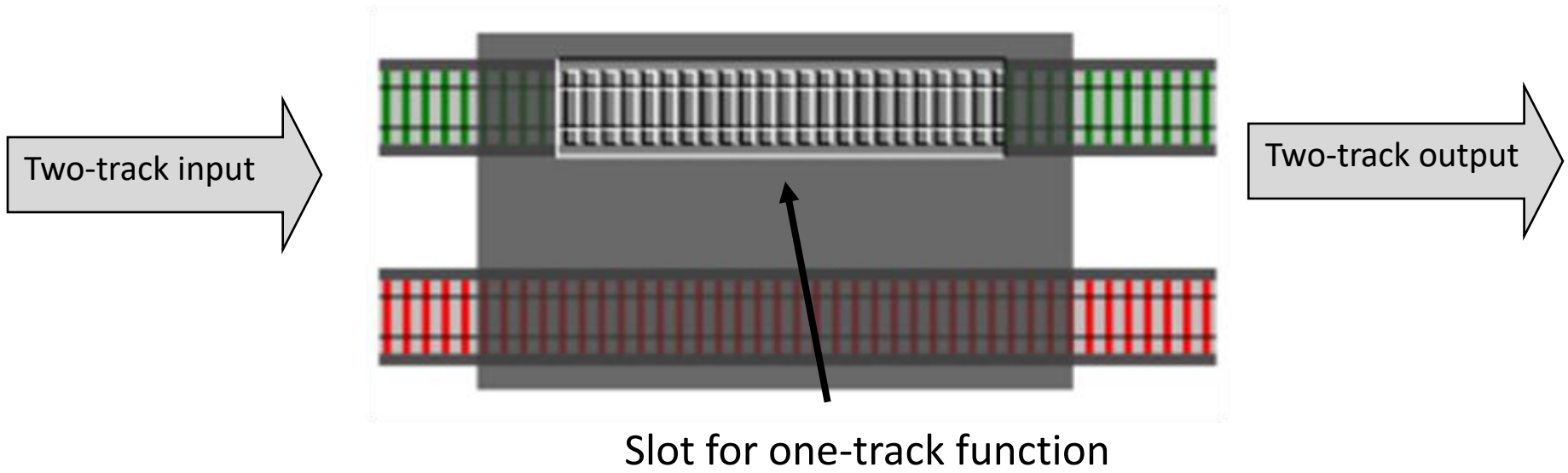


```
// trim spaces and lowercase  
let canonicalizeEmail input =  
  { input with email = input.email.Trim().ToLower() }
```

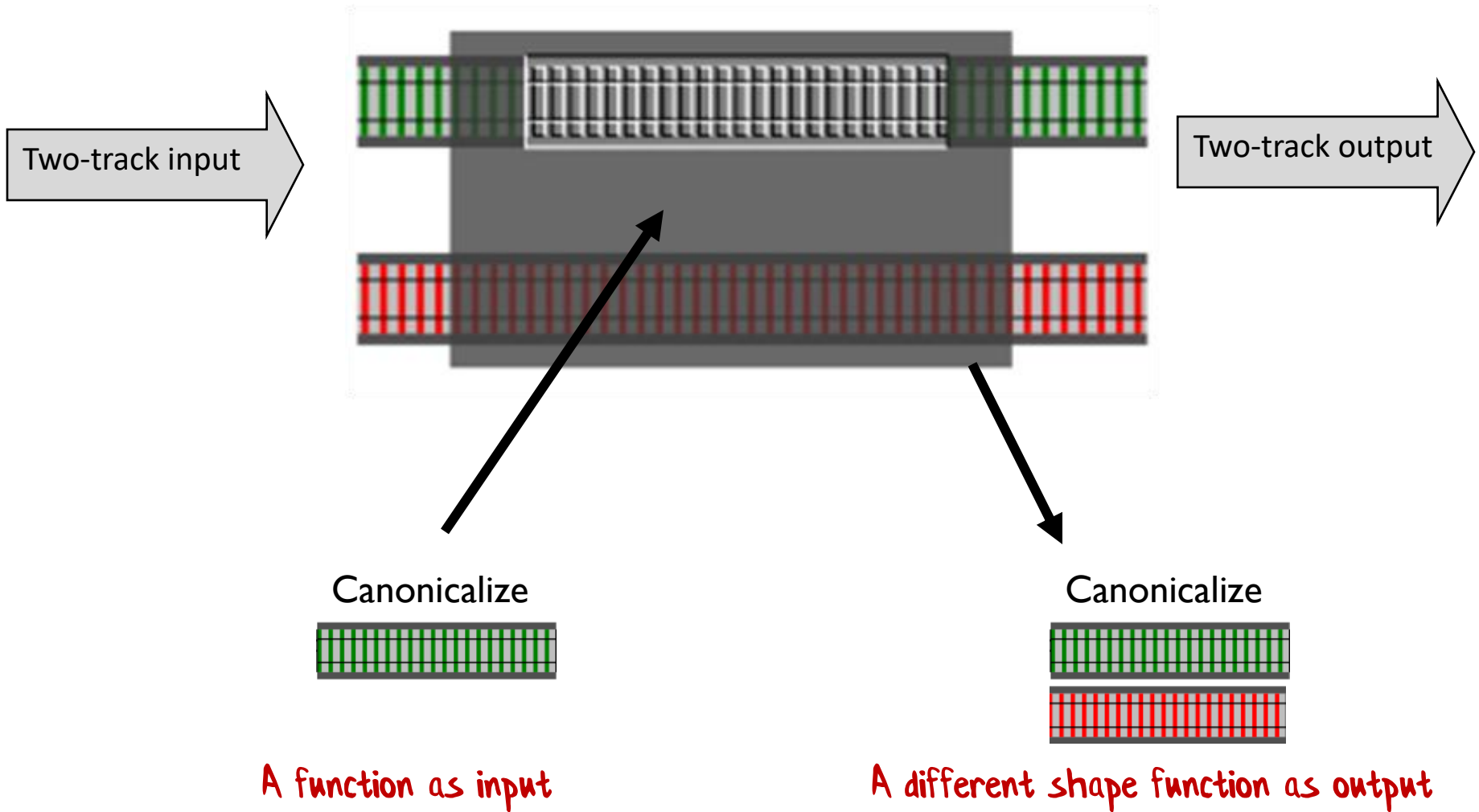
A simple function that doesn't generate errors – a "one-track" function.

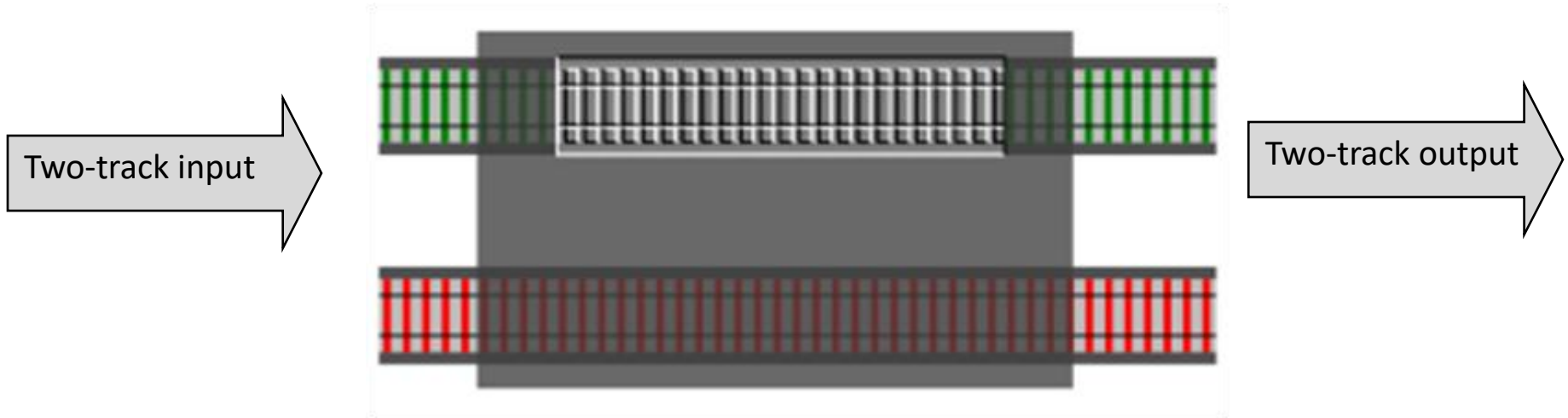


X
Won't compose



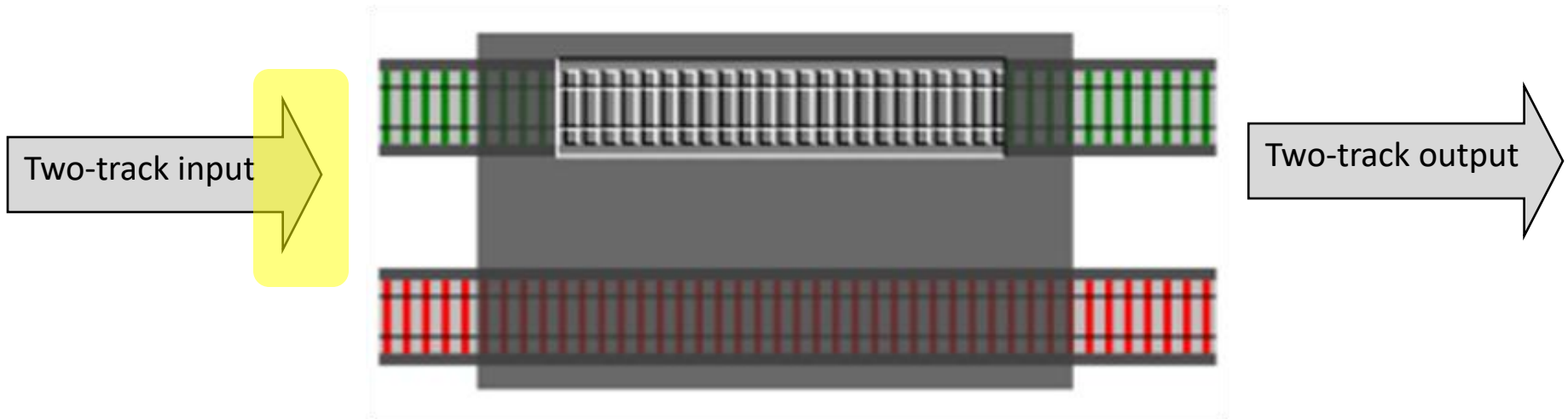
This adapter block is another "function transformer"



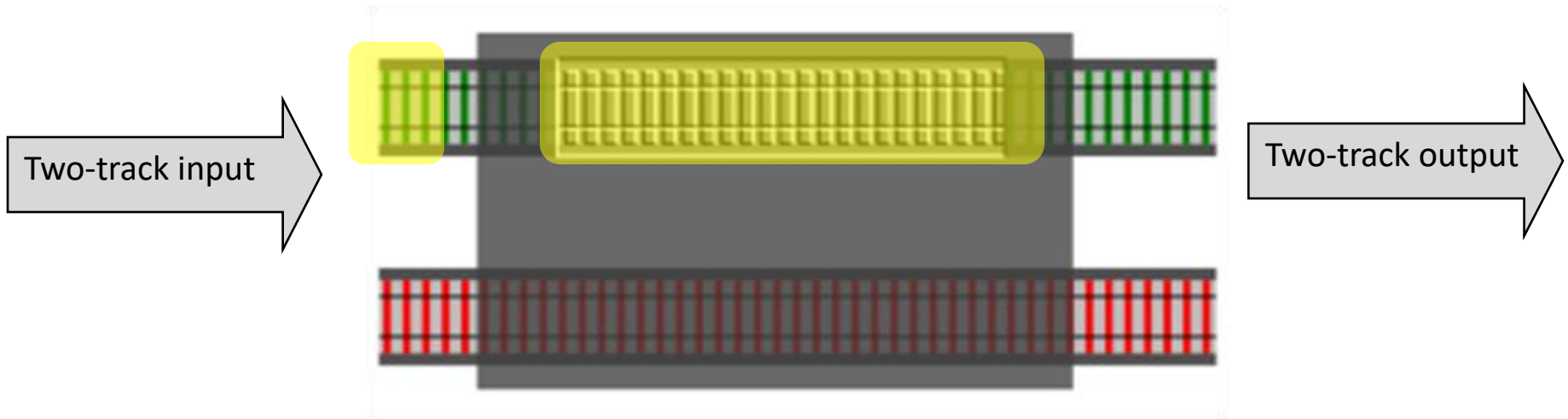


```
let map singleTrackFunction twoTrackInput =  
  match twoTrackInput with  
  | Ok s -> Ok (singleTrackFunction s)  
  | Error f -> Error f
```

*"map" is the common name for this kind of function.
In LINQ it's "Select"*

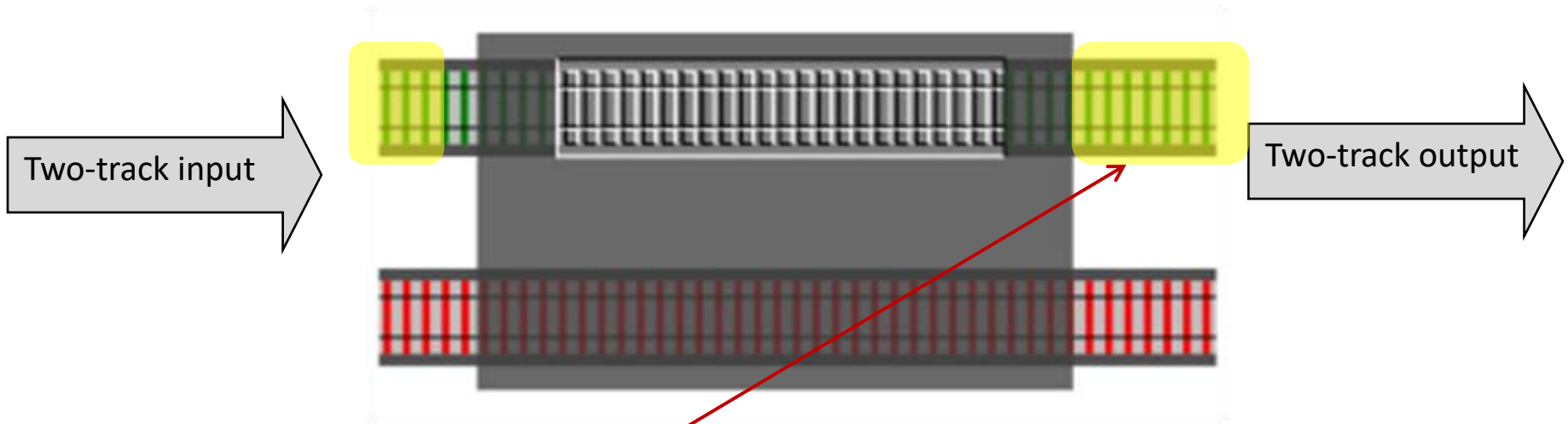


```
let map singleTrackFunction twoTrackInput =  
  match twoTrackInput with  
  | Ok s -> Ok (singleTrackFunction s)  
  | Error f -> Error f
```



```
let map singleTrackFunction twoTrackInput =  
  match twoTrackInput with  
  | Ok s -> Ok (singleTrackFunction s)  
  | Error f -> Error f
```

a) handle the Ok track

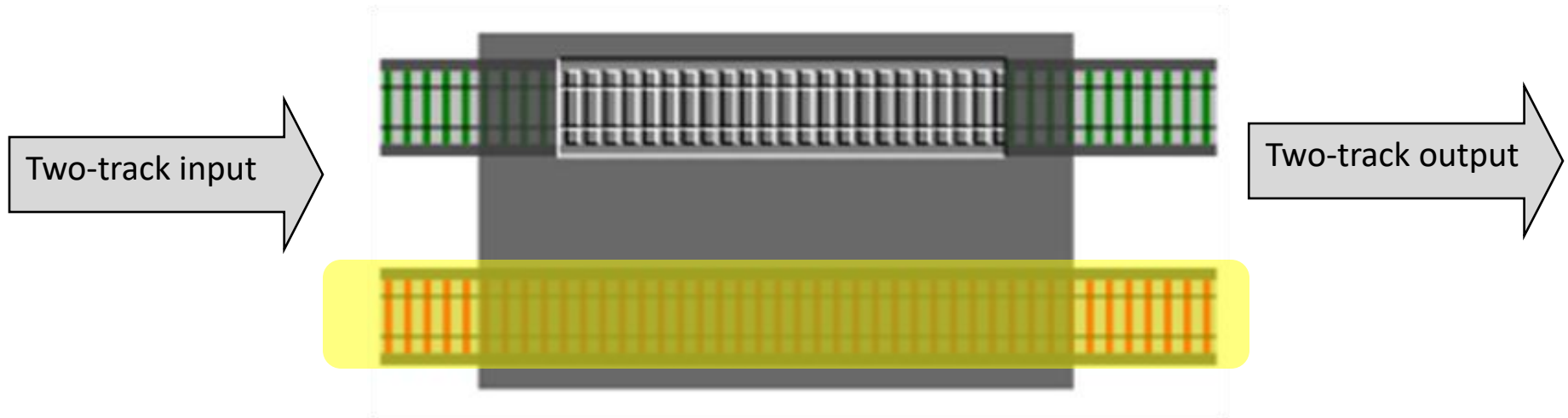


```

let map singleTrackFunction twoTrackInput =
  match twoTrackInput with
  | Ok s -> Ok (singleTrackFunction s)
  | Error f -> Error f

```

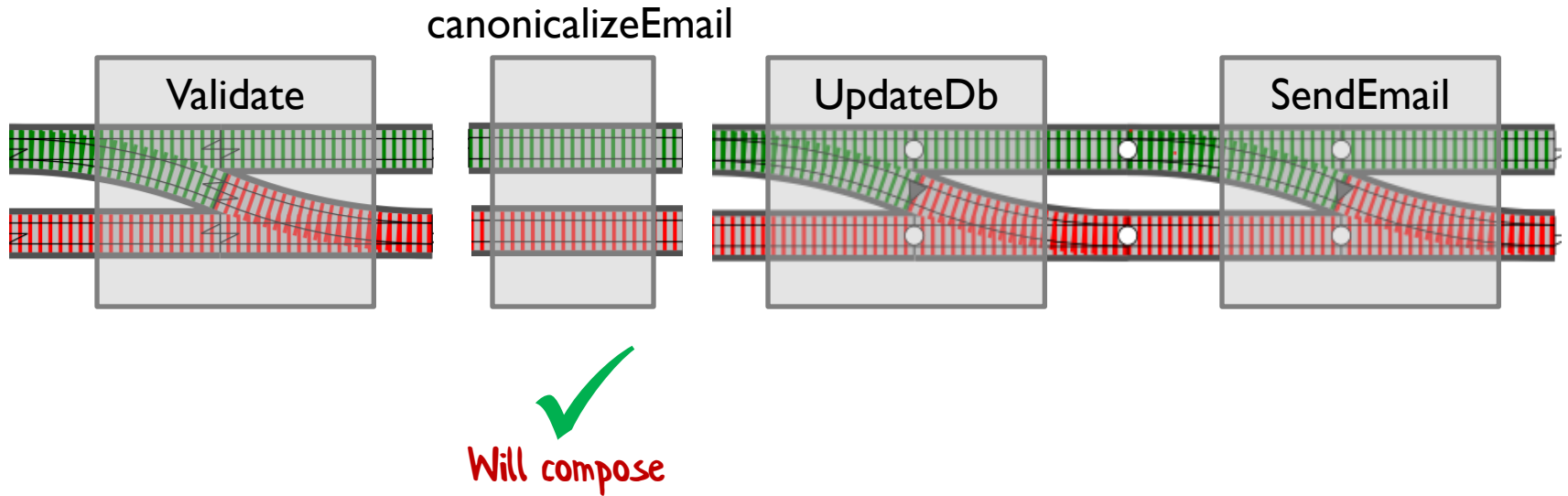
a) handle the Ok track



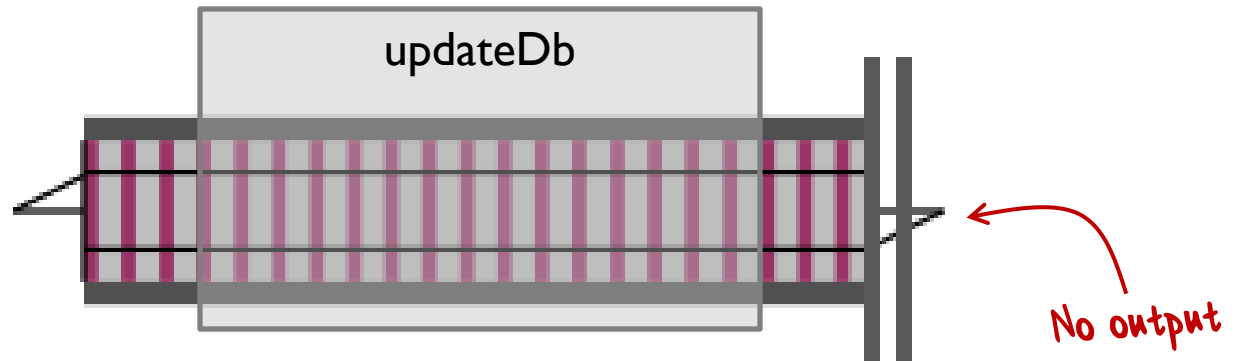
```
let map singleTrackFunction twoTrackInput =  
  match twoTrackInput with  
  | Ok s -> Ok (singleTrackFunction s)  
  | Error f -> Error f
```

b) Handle the Error track

Converting one-track functions

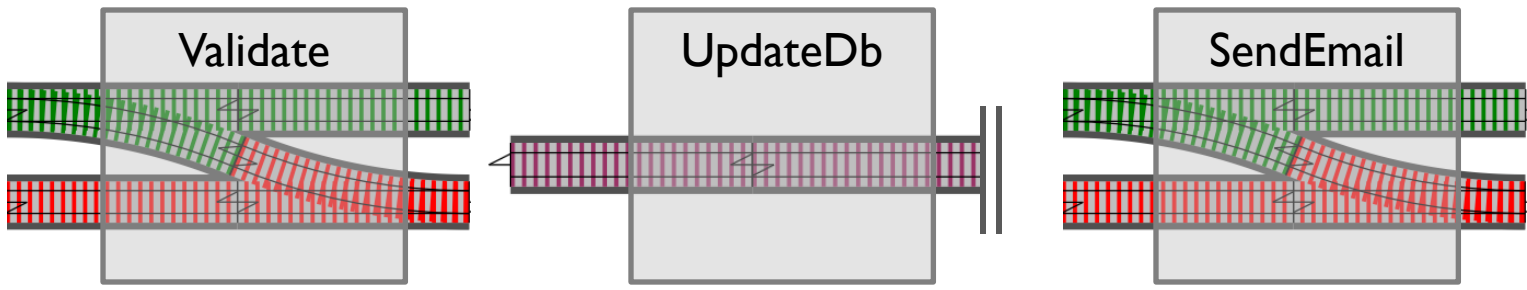


Working with "dead-end" functions

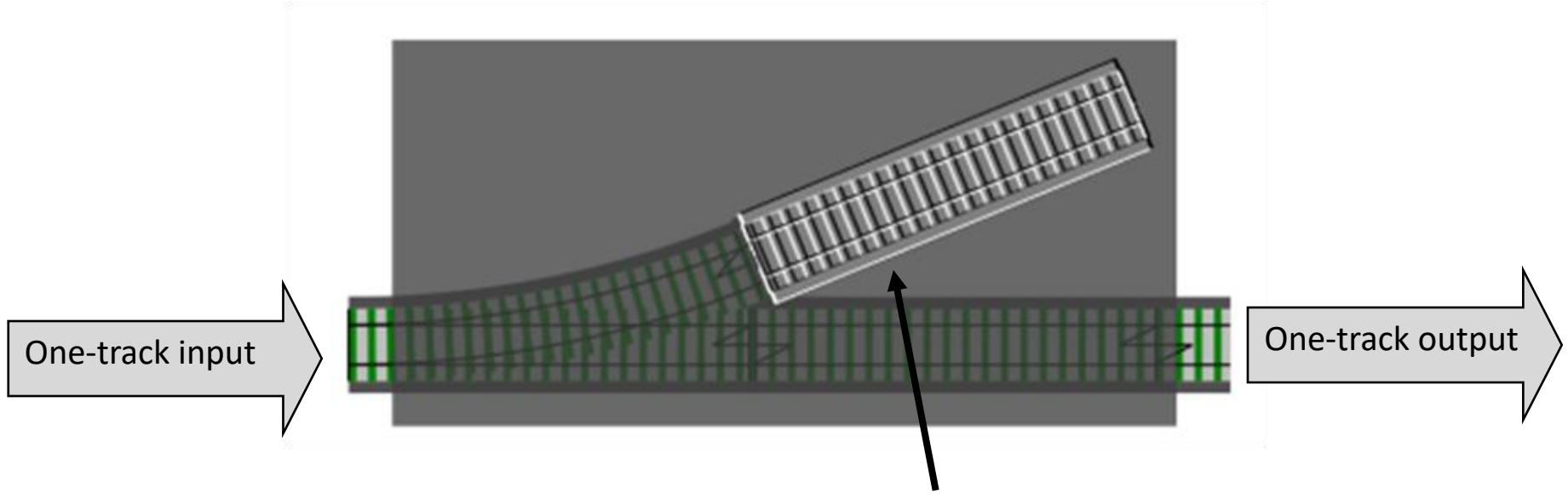


```
let updateDb request =  
  // do something  
  // return nothing at all
```

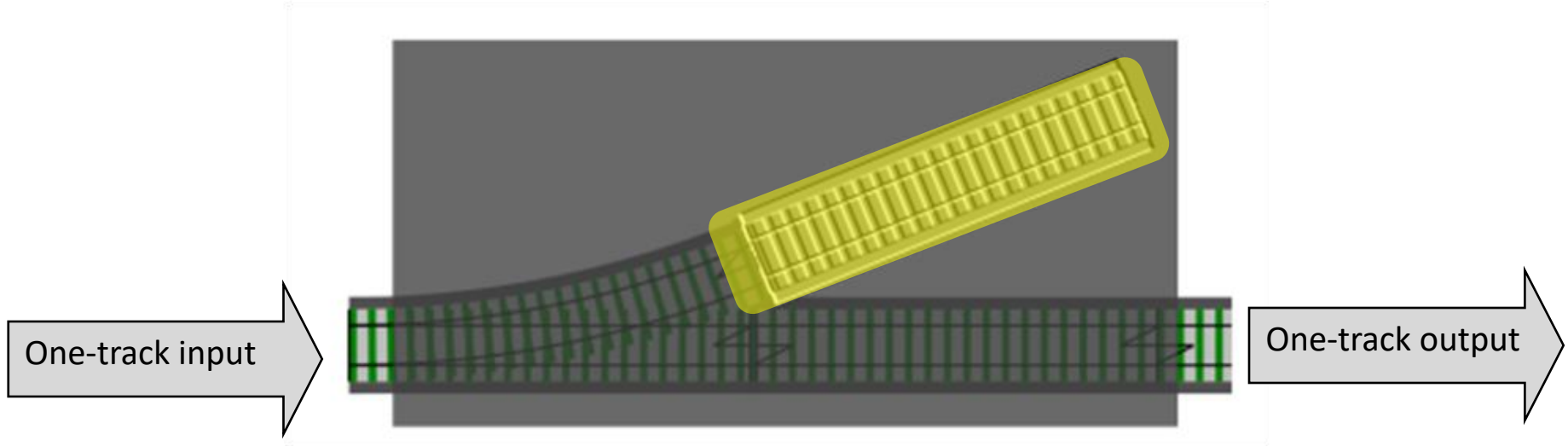
A function that doesn't return anything— a "dead-end" function.



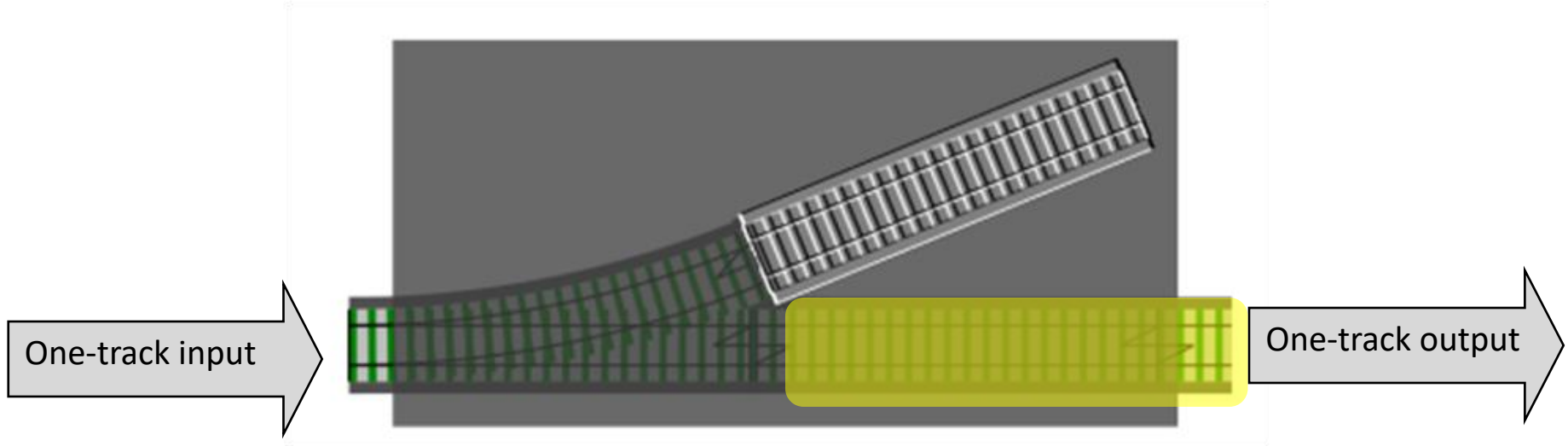
X
Won't compose



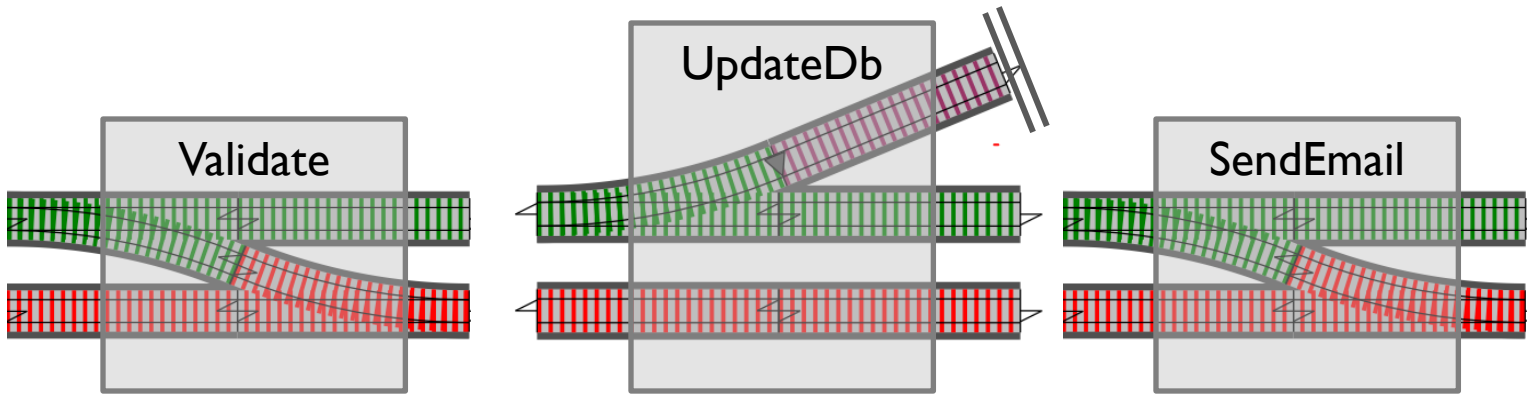
Slot for dead end function



```
let tee deadEndFunction oneTrackInput =  
  deadEndFunction oneTrackInput  
  oneTrackInput
```

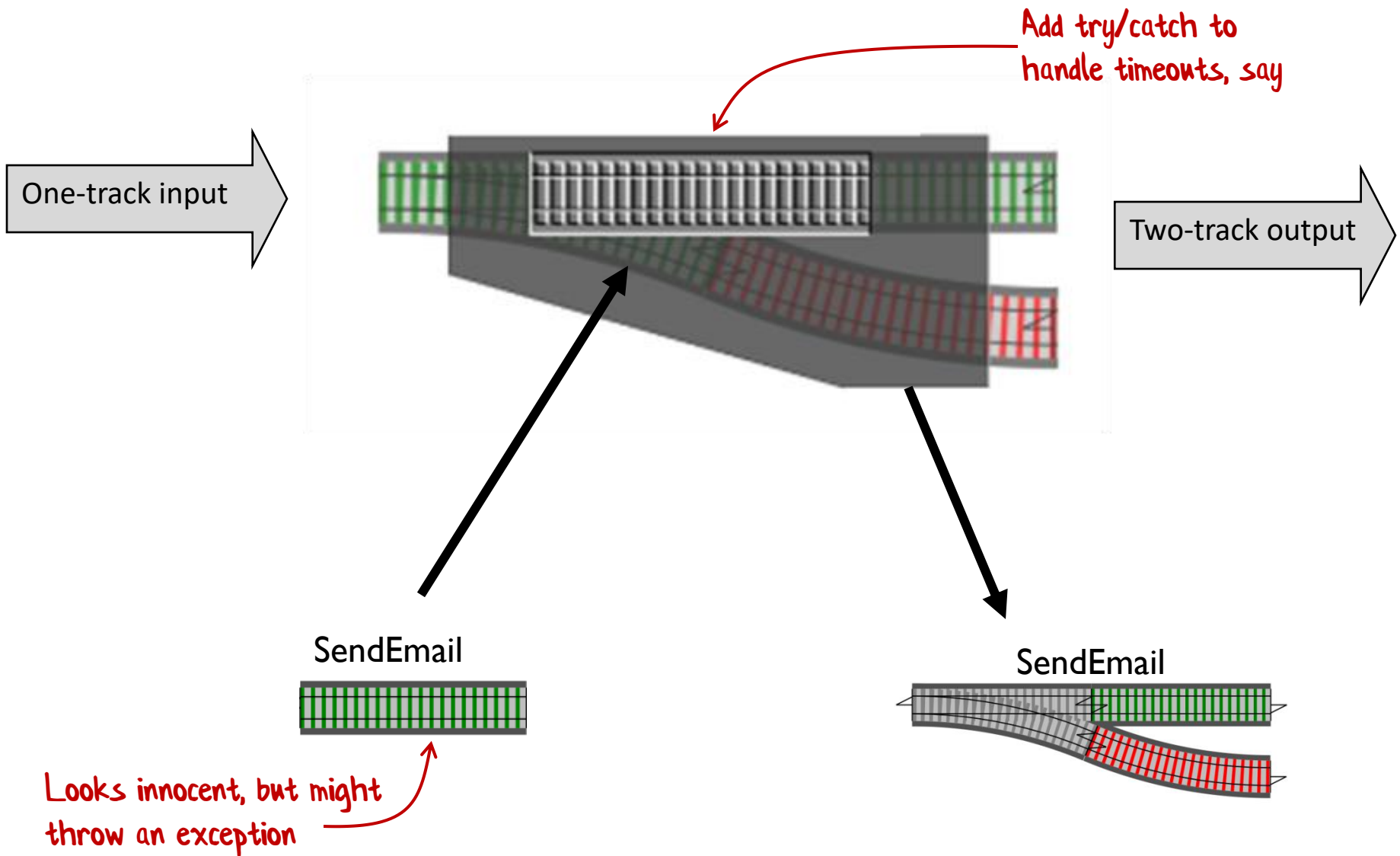


```
let tee deadEndFunction oneTrackInput =  
    deadEndFunction oneTrackInput  
    oneTrackInput
```



Will compose

Working with functions that throw exceptions



Looks innocent, but might throw an exception

Don't bother handling all possible exceptions:
E.g. FileNotFoundException, yes. OutOfMemory, no.

Guideline: Convert exceptions into Failures



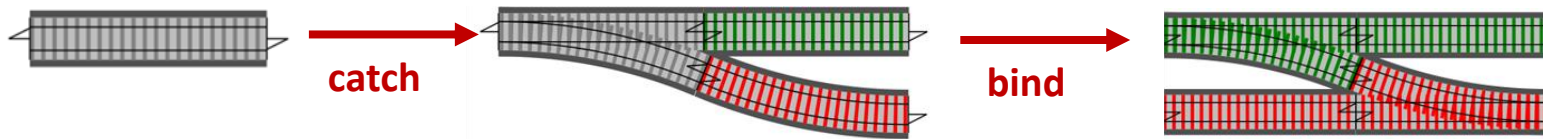
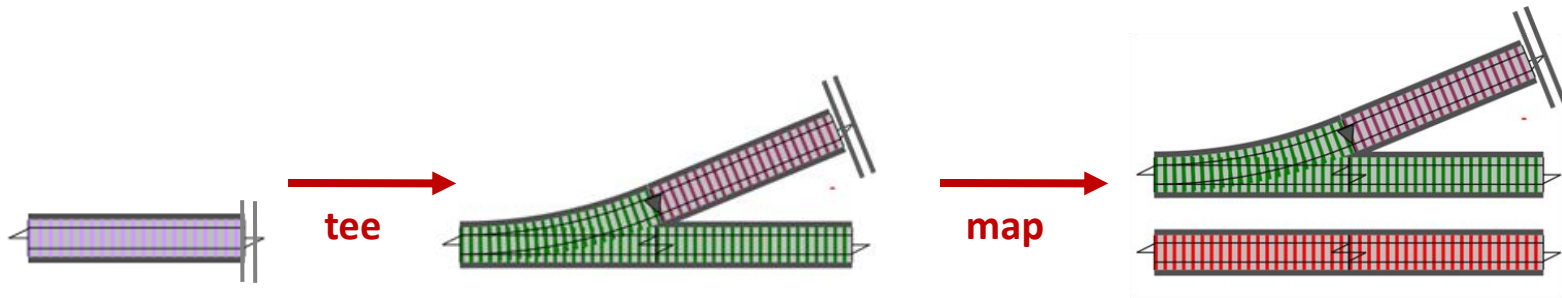
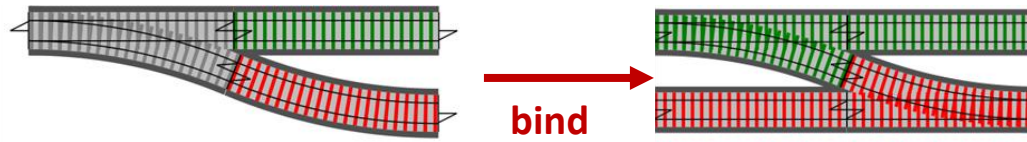
Even Yoda recommends not to use exception handling for control flow:

"Do or do not, there is no try".

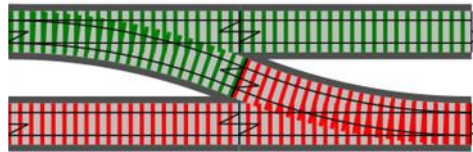
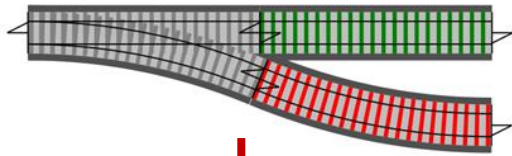
Functions won't compose?

Use "function transformers" to change their shape!

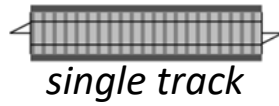
Here are the transformations we used so far



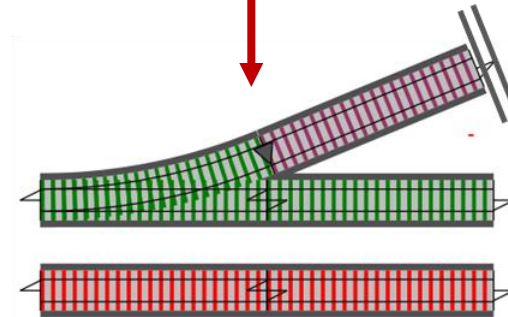
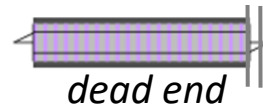
Validate



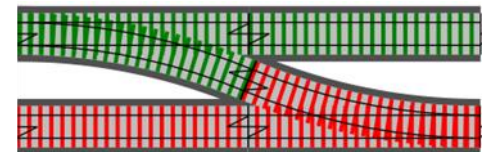
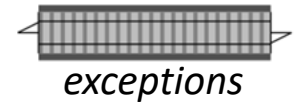
Canonicalize



UpdateDb



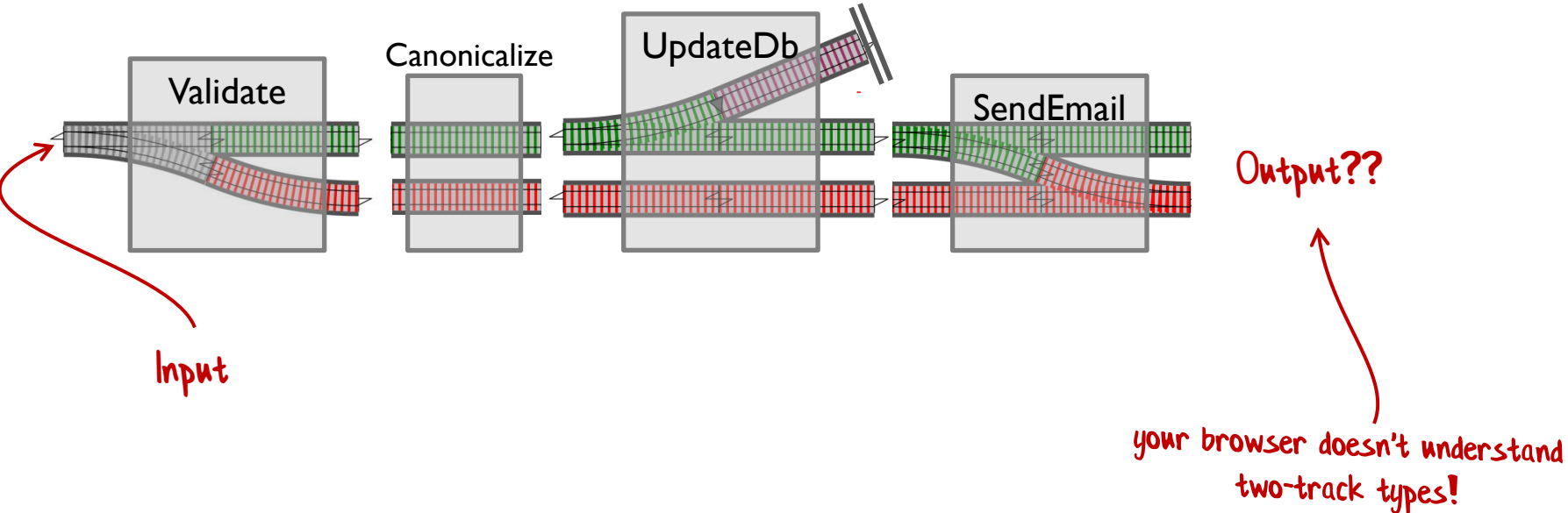
SendEmail



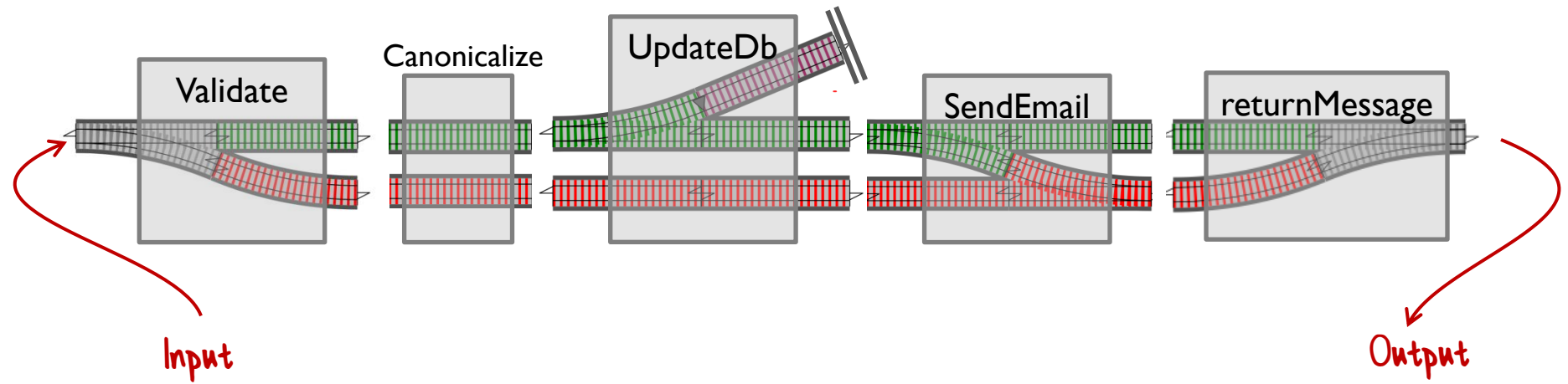
Now we *can* compose them easily!

Putting it all together
into a pipeline

Putting it all together



Putting it all together

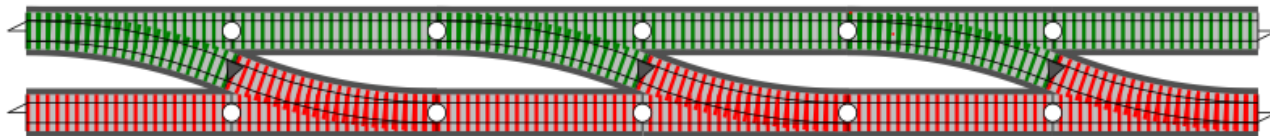


```
let returnMessage result =  
  match result with  
  | Ok obj -> "200" + obj.ToJson()  
  | Error errorMsg -> "400" + errorMsg
```

Demo #4:

The complete pipeline


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



As promised at the beginning!

Part V:

Designing error types

Unhappy paths are requirements too

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

```
// returns Result<Input, string> =
```

Using strings is not good



```
let validateInput input =  
  if input.name = "" then  
    Error NameMustNotBeBlank  
  else if input.email = "" then  
    Error EmailMustNotBeBlank  
  else  
    Ok input // happy path
```

```
type ErrorMessage =  
  | NameMustNotBeBlank  
  | EmailMustNotBeBlank
```

```
// returns Result<Input, ErrorMessage> =
```

*Defined a special type
rather than string*

```
let validateInput input =  
  if input.name = "" then  
    Error NameMustNotBeBlank  
  else if input.email = "" then  
    Error EmailMustNotBeBlank  
  else if (input.email doesn't match regex) then  
    Error EmailNotValid input.email  
  else  
    Ok input // happy path
```

Add invalid
email as data

```
type ErrorMessage =  
  | NameMustNotBeBlank  
  | EmailMustNotBeBlank  
  | EmailNotValid of EmailAddress
```

```
type ErrorMessage =  
  | NameMustNotBeBlank  
  | EmailMustNotBeBlank  
  | EmailNotValid of EmailAddress
```

```
type ErrorMessage =
  | NameMustNotBeBlank
  | EmailMustNotBeBlank
  | EmailNotValid of EmailAddress
  // database errors
  | UserIdNotValid of UserId
  | DbUserNotFoundError of UserId
  | DbTimeout of ConnectionString
  | DbConcurrencyError
  | DbAuthorizationError of ConnectionString * Credentials
  // SMTP errors
  | Smtptimeout of Smtptimeout
  | Smtpbadrecipient of EmailAddress
```

Documentation of everything
that can go wrong --

And it's type-safe
documentation that can't go
out of date!

* Credentials
Also triggers important
DDD conversations

Designing for errors - review

```
type ErrorMessage =  
  | NameMustNotBeBlank  
  | EmailMustNotBeBlank  
  | EmailNotValid of EmailAddress  
  // database errors  
  | UserIdNotValid of UserId  
  | DbUserNotFound of UserId  
  | DbTimeout of ConnectionString  
  | DbConcurrencyError  
  | DbAuthorizationError of ConnectionString * Credentials  
  // SMTP errors  
  | Smtptimeout of Smtptimeout  
  | Smtpbadrecipient of EmailAddress
```

Documentation of everything that
can go wrong.

Type-safe -- can't go out of date!

Surfaces hidden requirements.

Test against error codes,
not strings.

Makes translation easier.

Demo #5:

Pipeline with error type
and language translation

What errors should be modeled?

Three kinds of errors

- **Domain errors** are to be expected as part of the business process
- **Panics** leave the system in an unknown state
- **Infrastructure errors** are expected as part of the architecture

Design tip:
Be aware of different kinds of errors

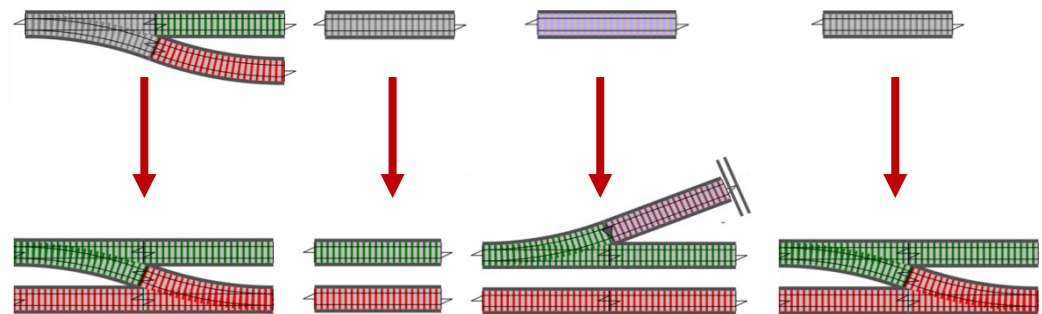
Don't always use Result for everything!

*I have a blog post called
"Against Railway-Oriented Programming" 😊*

Conclusion

Review of error handling in FP

- Errors are part of the domain model too!
- Use two-track error handling rather than exceptions
- Use "function transformers" to change the shape of a function so they can be composed

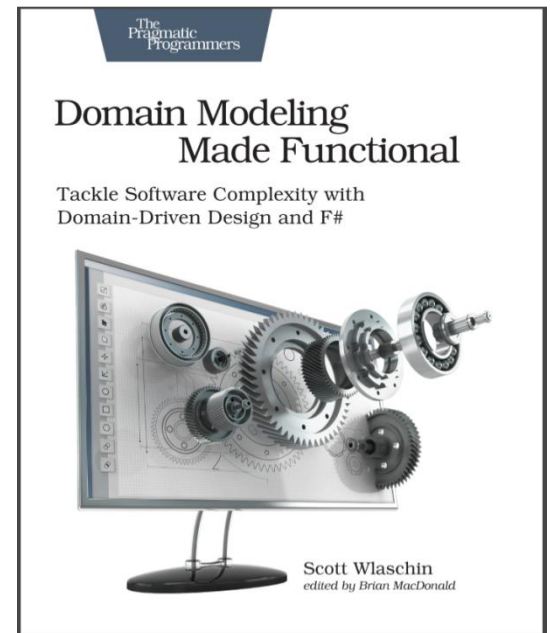


Thanks!

More at fsharpforfunandprofit.com/rop

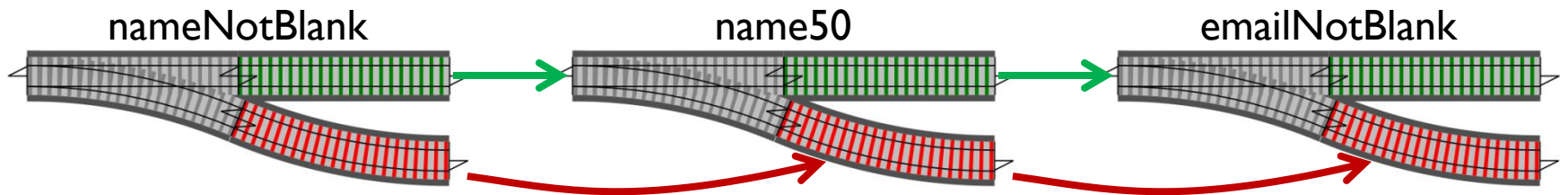
Code at <https://github.com/swlaschin/TechTrain2021>

My book!



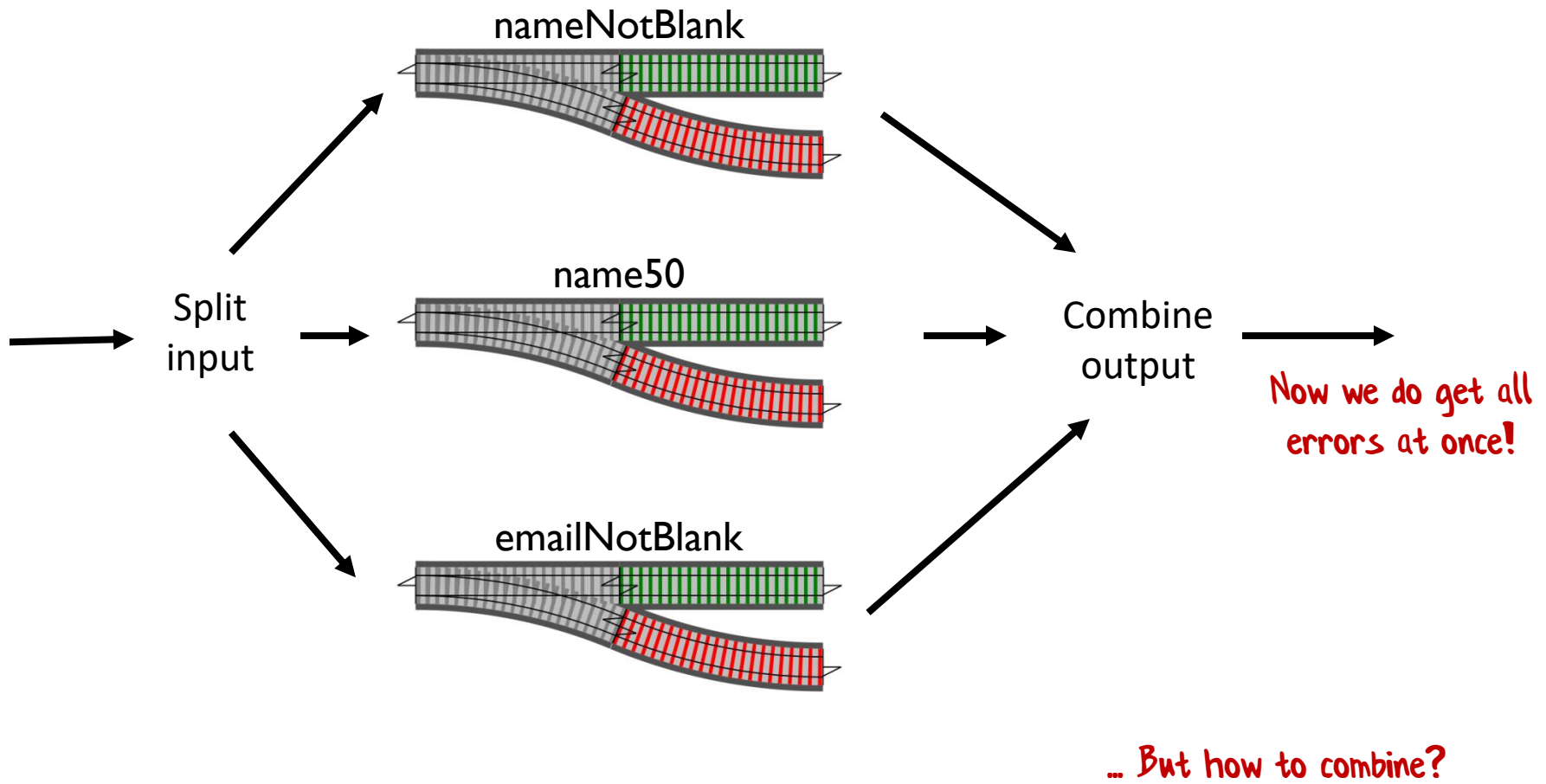
Bonus! Part VI:

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.



How can we combine errors?

```
type PersonalName = {  
  First: String10  
  Last: String10  
}
```

A record to be constructed.
It uses constrained types



```
let firstOrError = validateFirst first  
  // Validation<First, _>  
let lastOrError = validateLast last  
  // Validation<Last, _>
```

These values are
NOT normal values



How can we combine errors?

```
type PersonalName = {  
  First: String10  
  Last: String10  
}
```

```
let firstOrError = validateFirst first  
  // Validation<First,_>  
let lastOrError = validateLast last  
  // Validation<Last,_>
```

```
let name = {  
  First = firstOrError  
  Last = lastOrError  
}
```

We will get a compiler error
trying to assign them 😞




A different approach

```
let firstOrError = validateFirst first
  // Validation<First,_>
let lastOrError = validateLast last
  // Validation<Last,_>
```

```
// make a constructor for the record
```

```
let createPerson first last =
  {First=first; Last=last}
```

These parameters
are normal values



```
let personOrError =
```

```
  createPerson firstOrError lastOrError
```

We still get a compiler error when
passing validation values in 😞



A different approach, fixed!

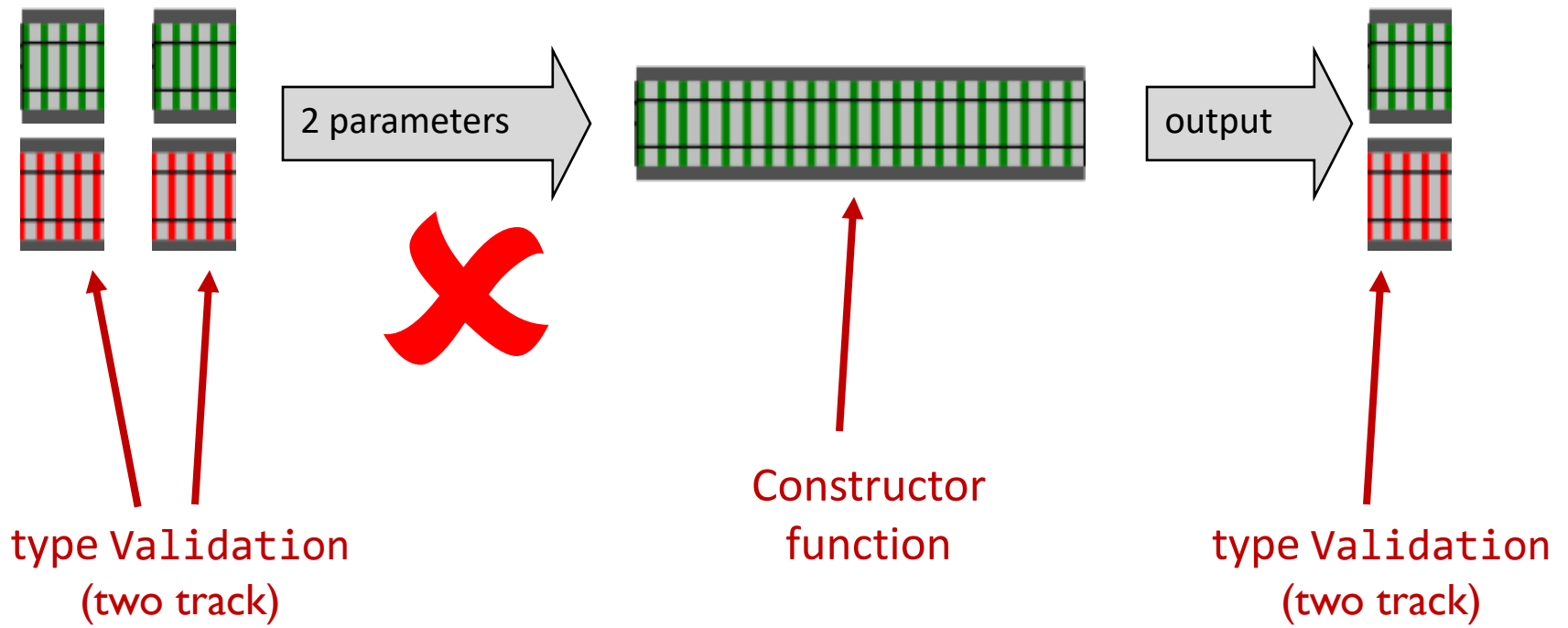
```
let firstOrError = validateFirst first
  // Validation<First,_>
let lastOrError = validateLast last
  // Validation<Last,_>

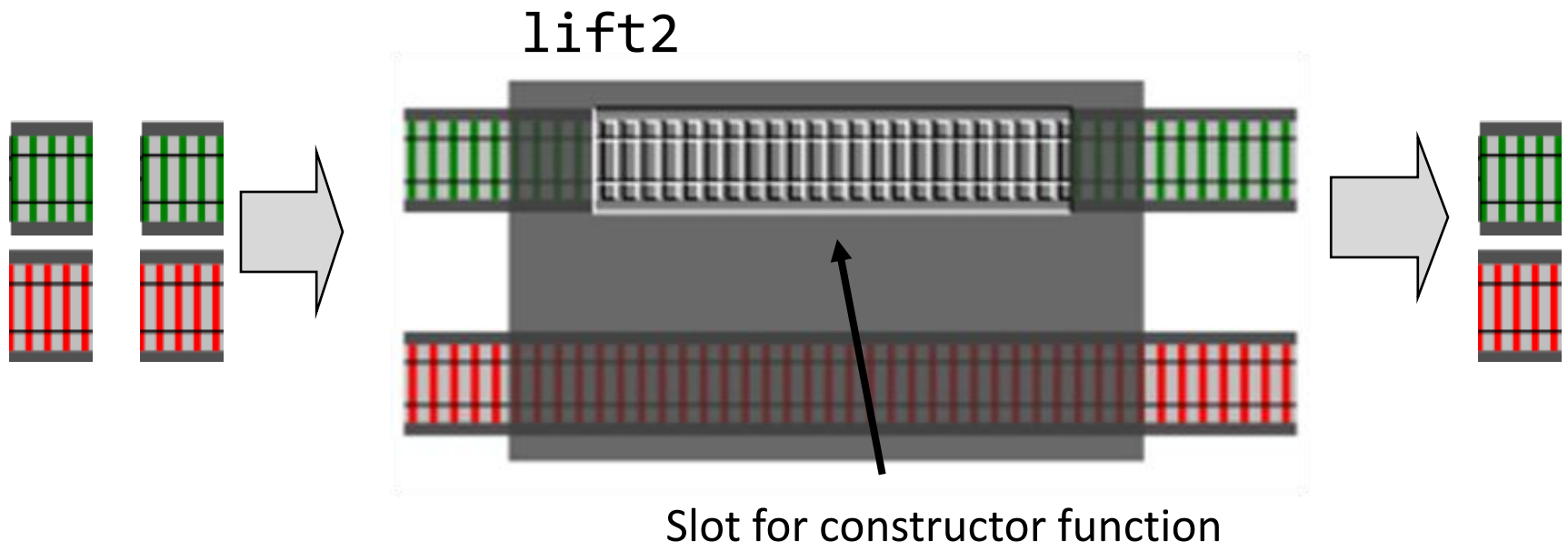
// make a constructor for the record
let createPerson first last =
  {First=first; Last=last}

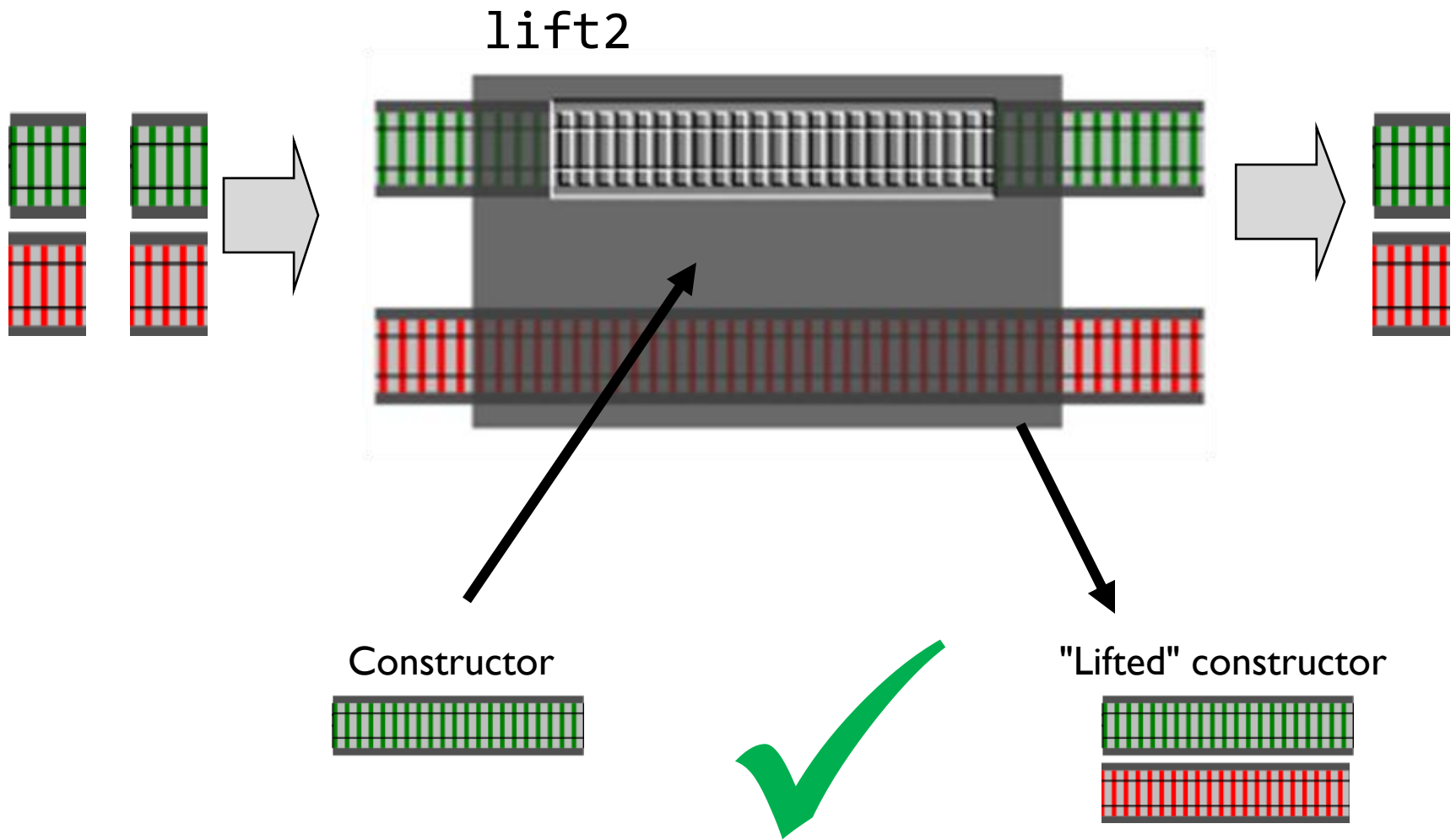
let personOrError =
  (lift2 createPerson) firstOrError lastOrError
  // returns Validation<Person,_>
```

Use "lift" to fix
the mismatch









The "recipe" for validation

You have a bunch of `Validation` errors, so

- a) Define a constructor for your record
- b) Use `lift2/lift3/lift4` functions to "adapt" the constructor to handle the validation results

Demo #6: Validation

End