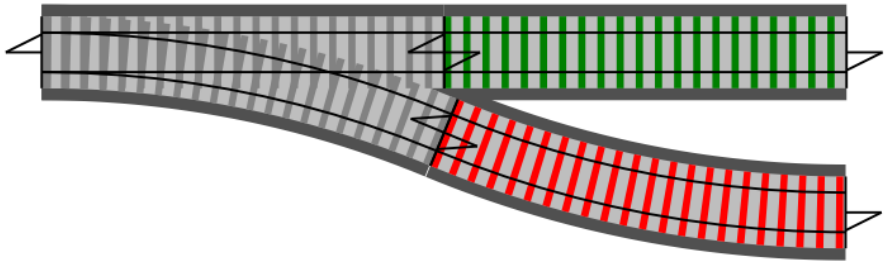
Modelling errors

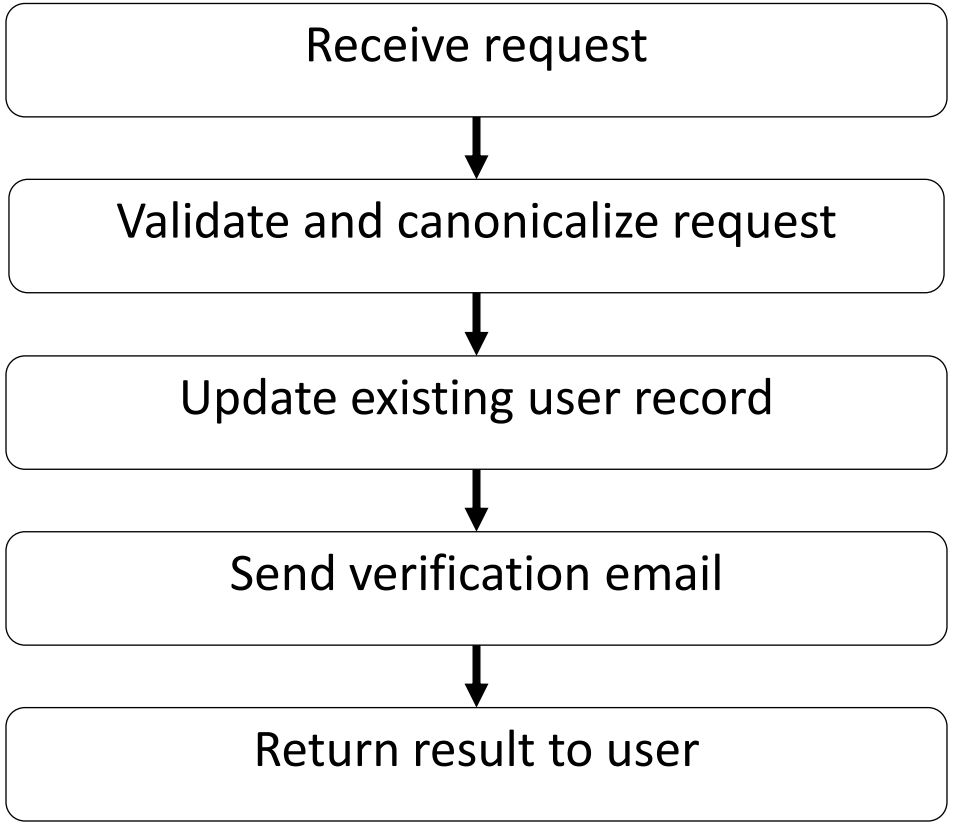


What do railways have to do with programming?

Happy path programming

Implementing a simple use case

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

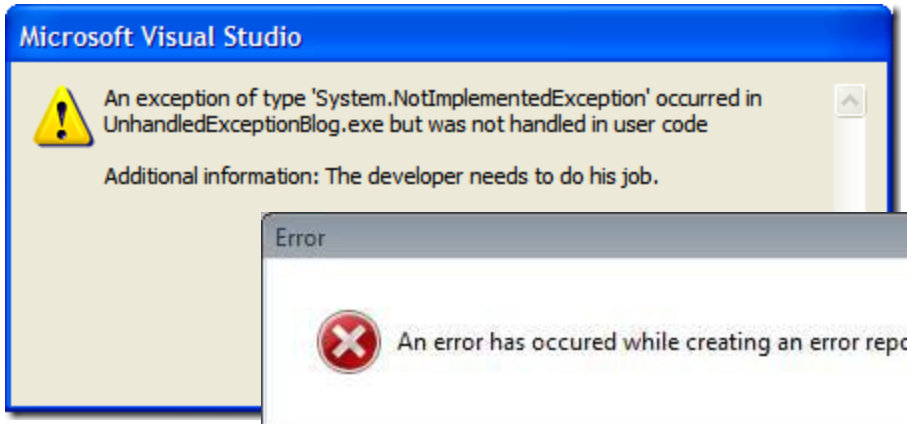



```
type Request = {
  userId: int;
  name: string;
  email: string }
```

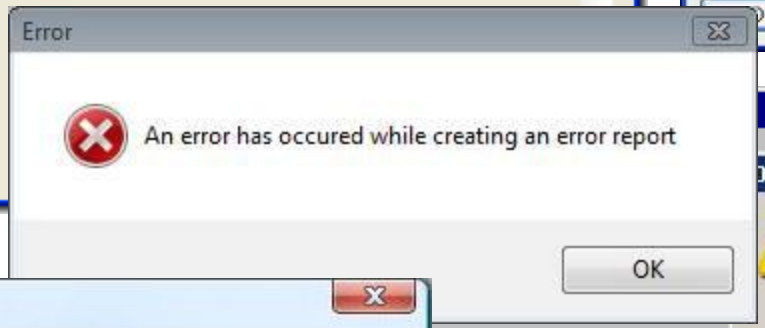
Straying from the happy path...

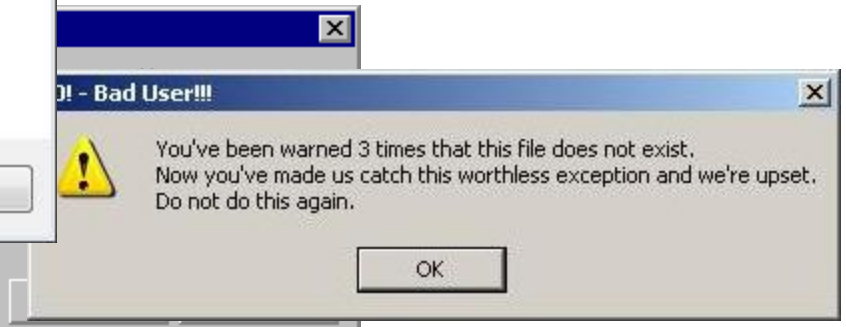
What do you do when
something goes wrong?

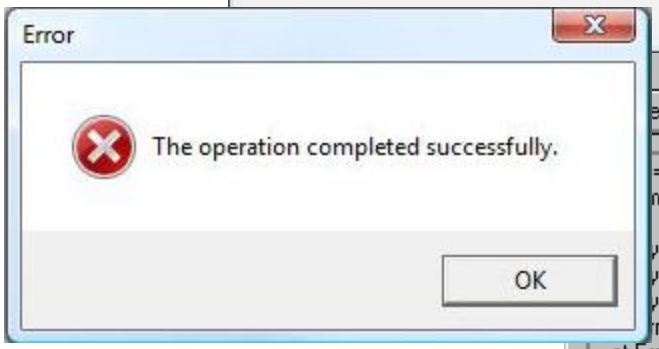

HAPPY PATH?
WHAT COULD POSSIBLY GO WRONG?

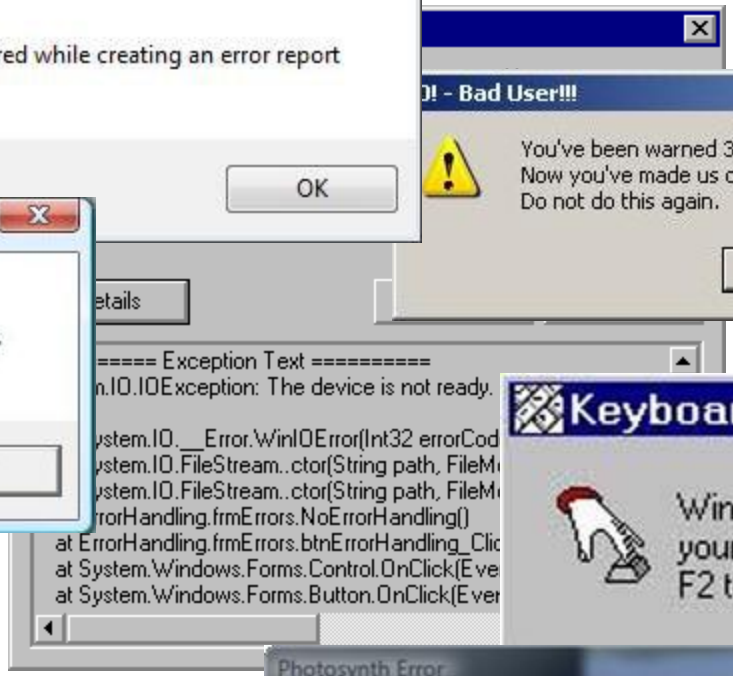

Microsoft Visual Studio
An exception of type 'System.NotImplementedException' occurred in UnhandledExceptionBlog.exe but was not handled in user code
Additional information: The developer needs to do his job.


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

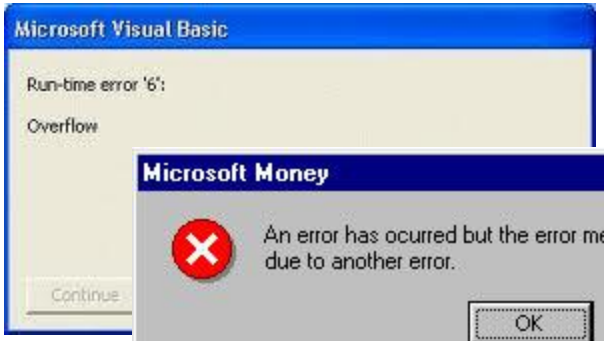

Error
An error has occured while creating an error report
OK


! - 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

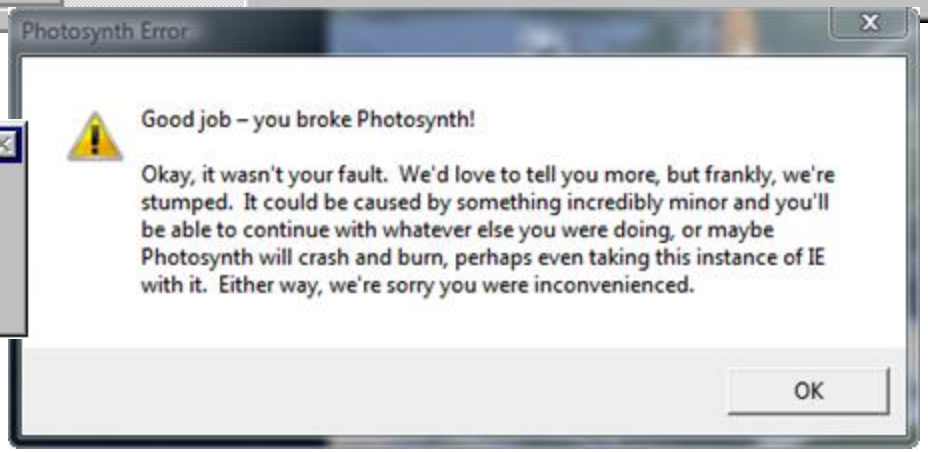

Error
The operation completed successfully.
OK


etails
===== Exception Text ==========
n.IO.IOException: The device is not ready.
ystem.IO.__Error.WinIOError(Int32 errorCod
ystem.IO.FileStream..ctor(String path, FileM
ystem.IO.FileStream..ctor(String path, FileM
rrorHandling.frmErrors.NoErrorHandling()
at ErrorHandling.frmErrors.btnErrorHandling_Clic
at System.Windows.Forms.Control.OnClick(Eve
at System.Windows.Forms.Button.OnClick(Ever


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


Microsoft Money
An error has ocurred but the error message cannot be retrieved due to another error.
OK


Photosynth Error
Good job – you broke Photosynth!
Okay, it wasn't your fault. We'd love to tell you more, but frankly, we're stumped. It could be caused by something incredibly minor and you'll be able to continue with whatever else you were doing, or maybe Photosynth will crash and burn, perhaps even taking this instance of IE with it. Either way, we're sorry you were inconvenienced.
OK

"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 }
```

Receive request

↓

Validate and canonicalize request → Name is blank, Email not valid

↓

Update existing user record → User not found, Db error

↓

Send verification email → Authorization error, Timeout

↓

Return result to user

```
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";
}
```

Q: What is the functional equivalent of this code?

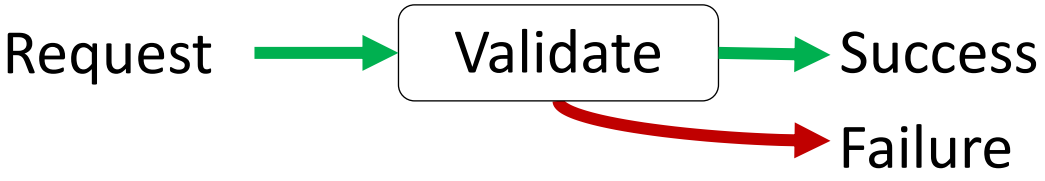
... and can we preserve the elegance of the original functional version?

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

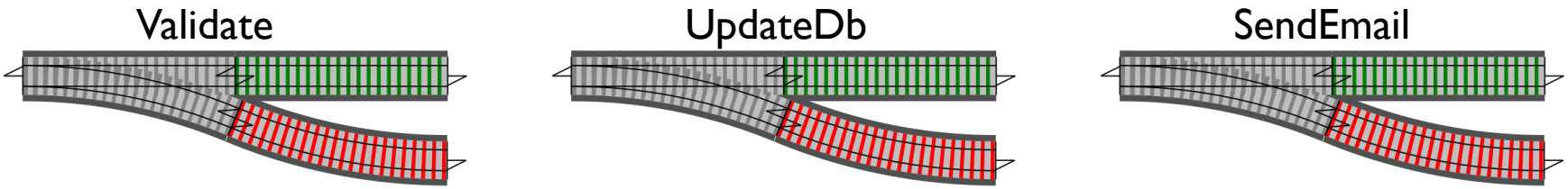
Use a *Result* type
for error handling

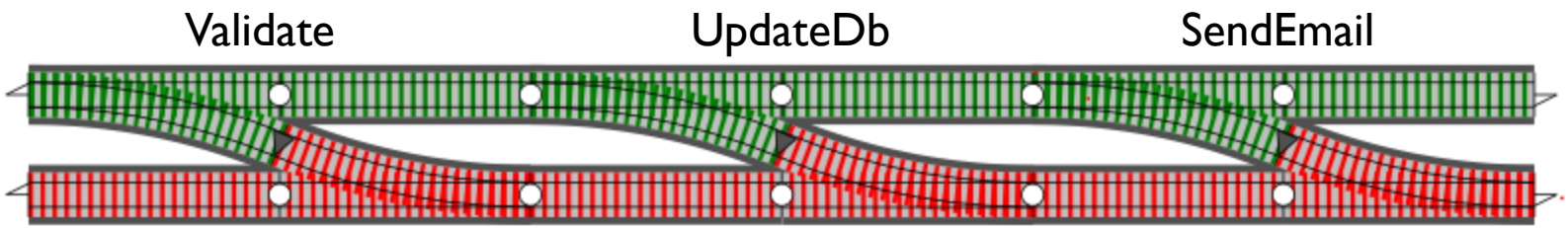

```
type Result =
    | Ok of SuccessValue
    | Error of ErrorValue
```

Define a choice type




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


Validate
UpdateDb
SendEmail



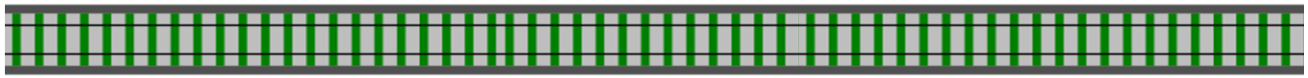
This is the "two track" model –
the basis for the "Railway Oriented Programming"
approach to error handling.

Functional flow without error handling

Before

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

One track

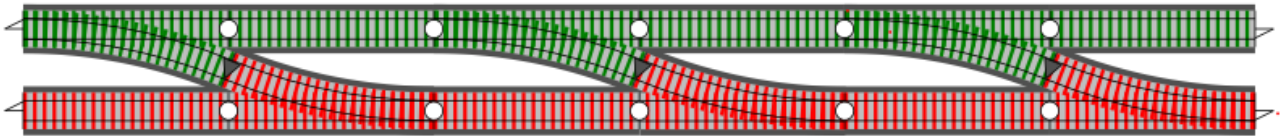


Functional flow with error handling

After

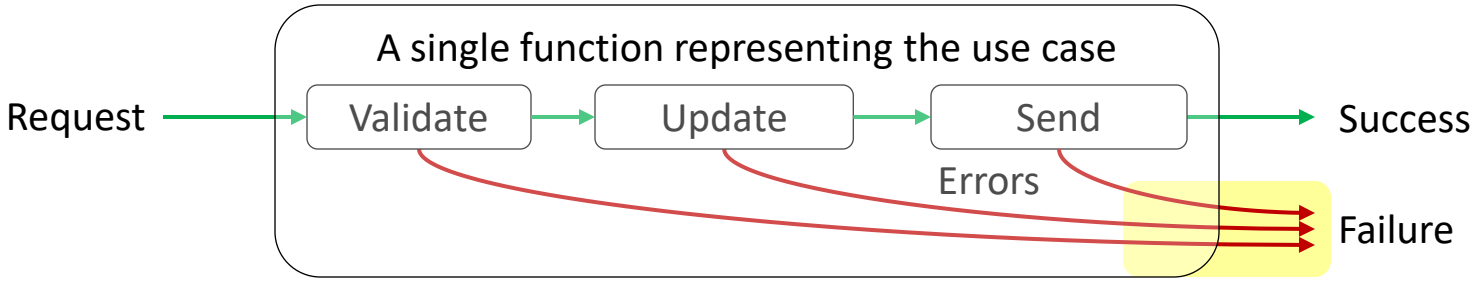
```
let updateCustomerWithErrorHandling =
  receiveRequest()
  |> validateRequest
  |> canonicalizeEmail
  |> updateDbFromRequest
  |> sendEmail
  |> returnMessage
```

Two track



See fsharpforfunandprofit.com/rop

Designing the unhappy path



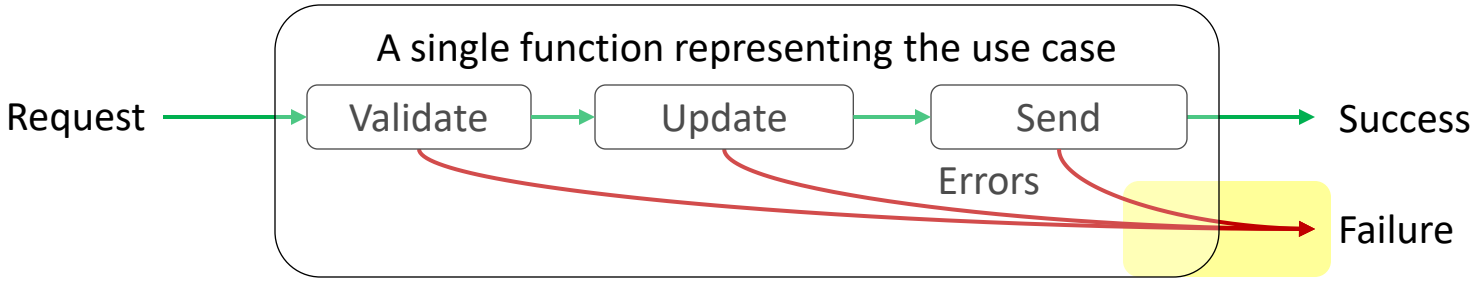
How can a function have more than one output?

Use a choice type!

```
type Result =
    | Ok
    | ValidationError
    | UpdateError
    | SmtpError
```

But maybe too specific for this case?

Functional design

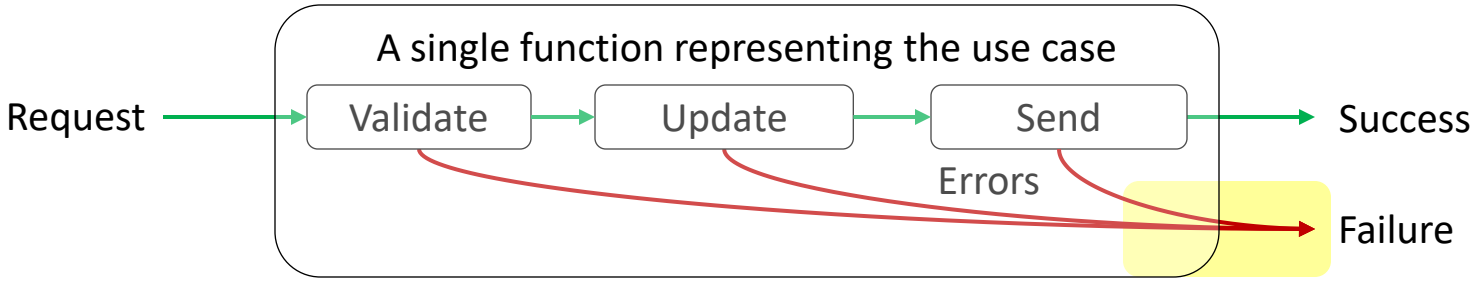


How can a function have more than one output?

```
type Result =
    | Ok
    | Error
```

Much more generic – but no data!

Functional design



How can a function have more than one output?

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

This is just what we want

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
```

Documentation of everything that can go wrong --

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

Also triggers important DDD conversations

Designing for errors - review

```
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 SmtpConnection
  | SmtpBadRecipient of EmailAddress
```

Documentation of everything that can go wrong.

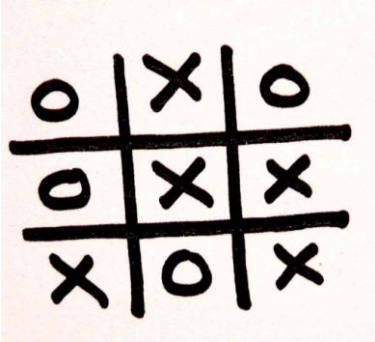
Type-safe -- can't go out of date!

Surfaces hidden requirements.

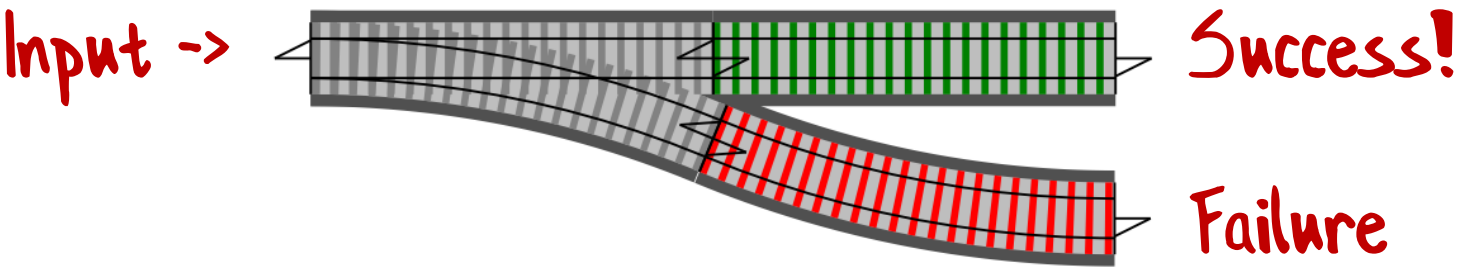
Test against error codes, not strings.

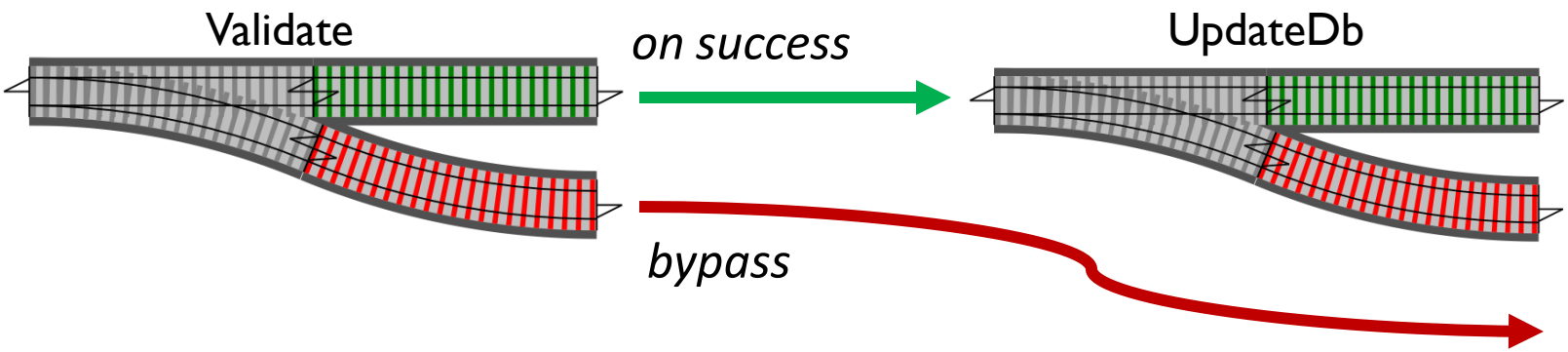
Makes translation easier.

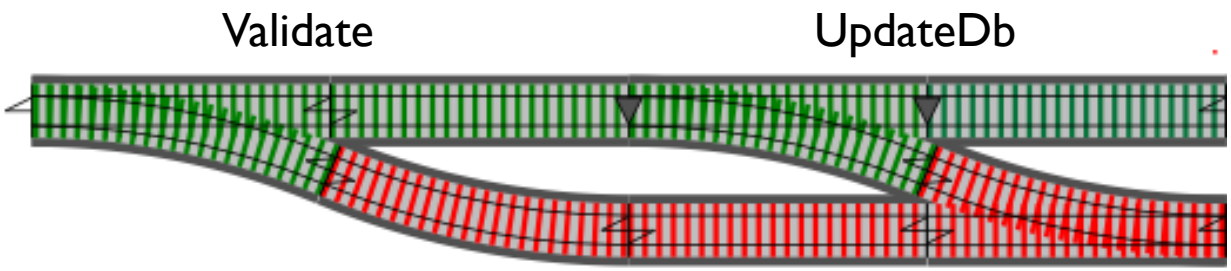
Exercise:
Add errors to the domain models

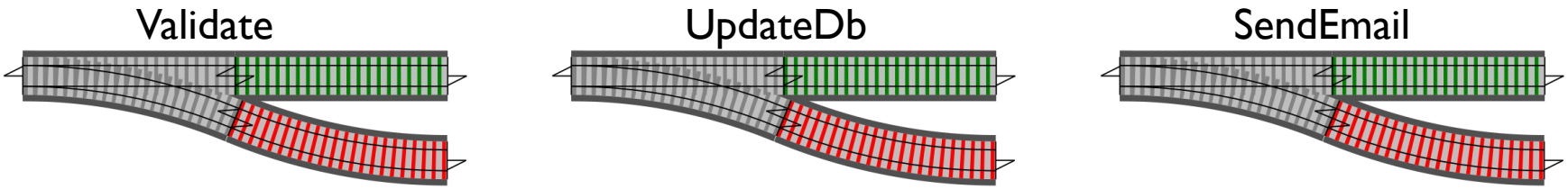


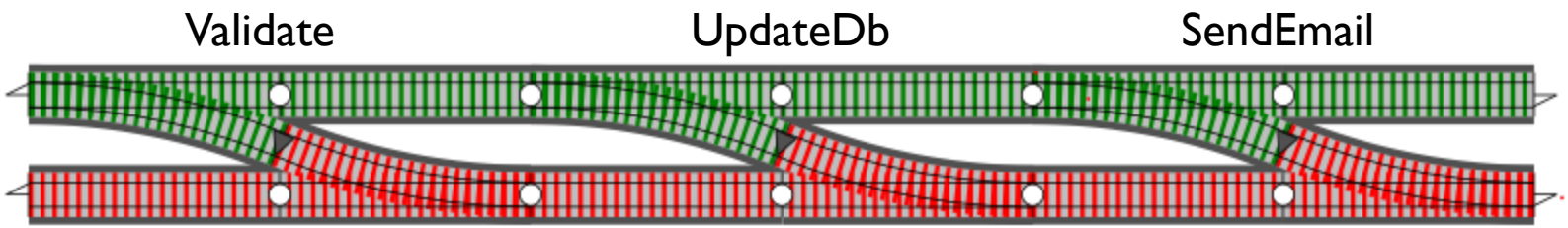
How to implement Railway Oriented Programming


Input ->
Success!
Failure


Validate
on success
UpdateDb
bypass

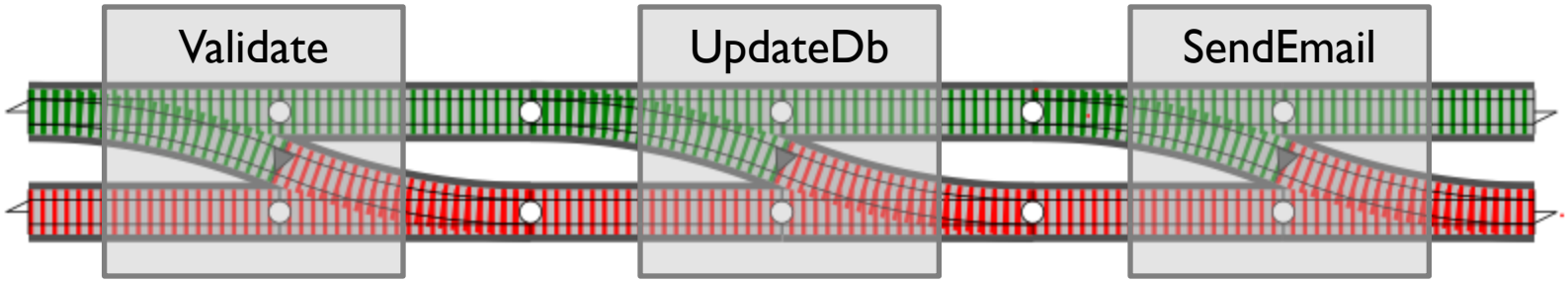

Validate
UpdateDb


Validate
UpdateDb
SendEmail


Validate
UpdateDb
SendEmail

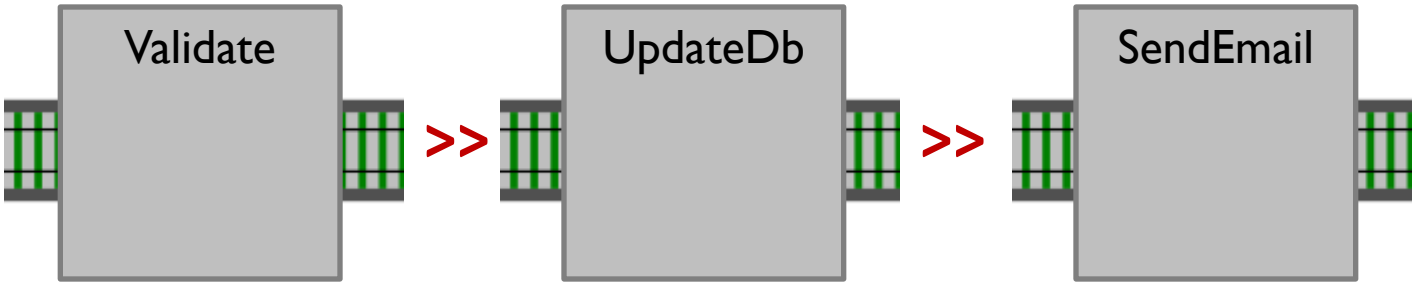
How to compose
these functions?

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

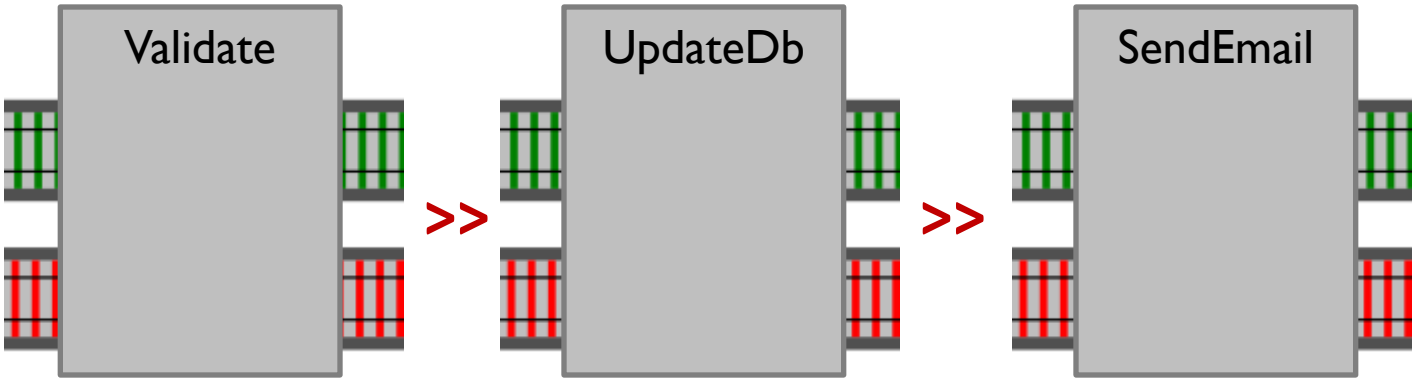


Here we have a series of black box functions
that are straddling a two-track railway.

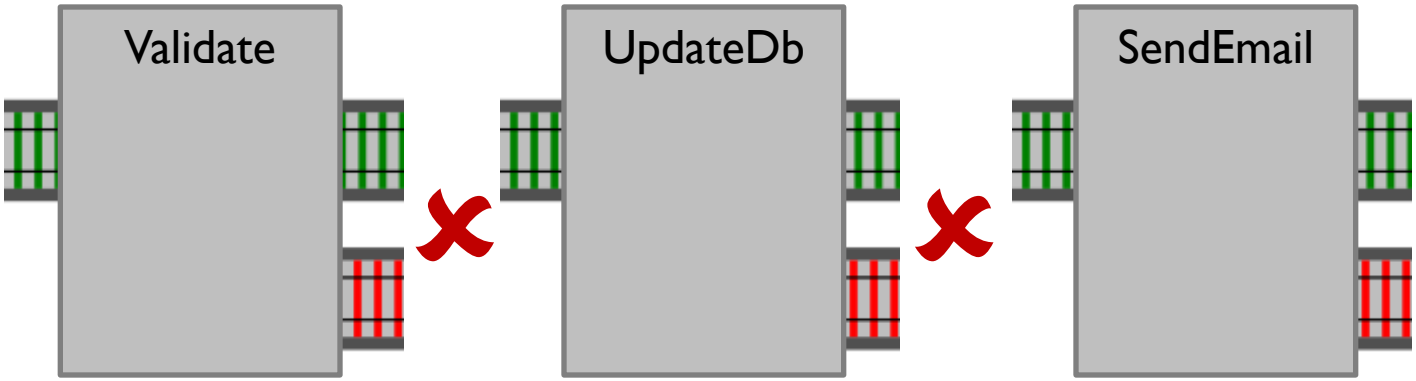
Inside each box there is a switch function.



Composing one-track functions is fine...



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

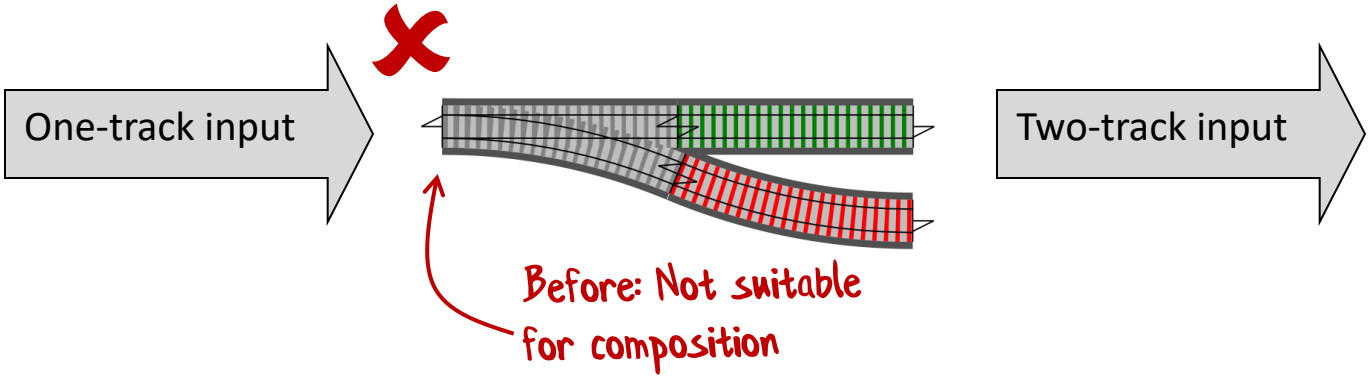


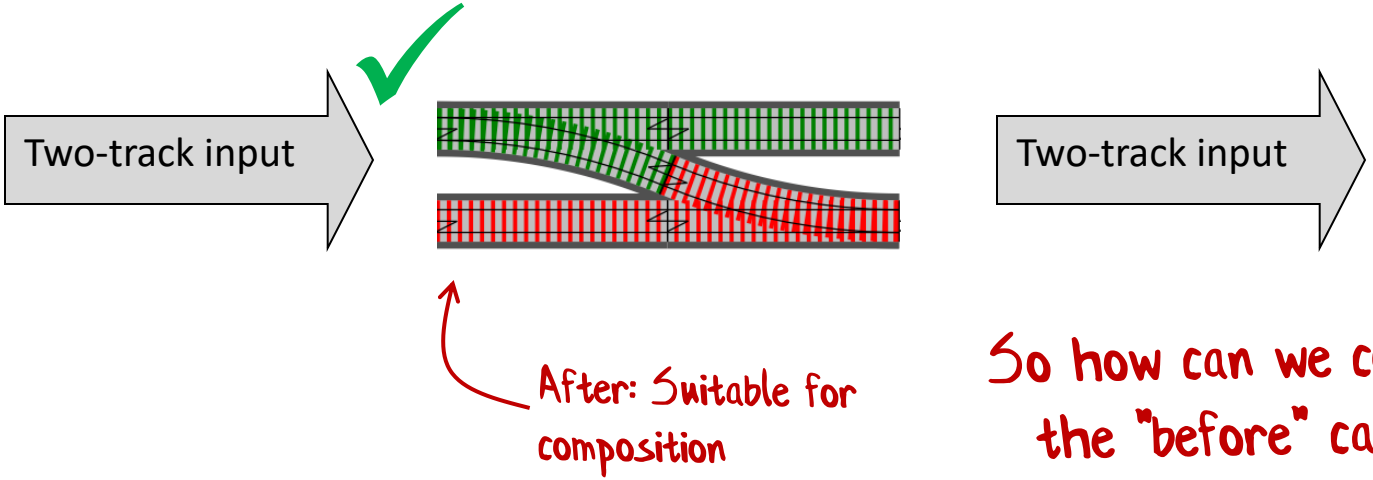
... but composing switches is not allowed!

How to combine the mismatched functions?

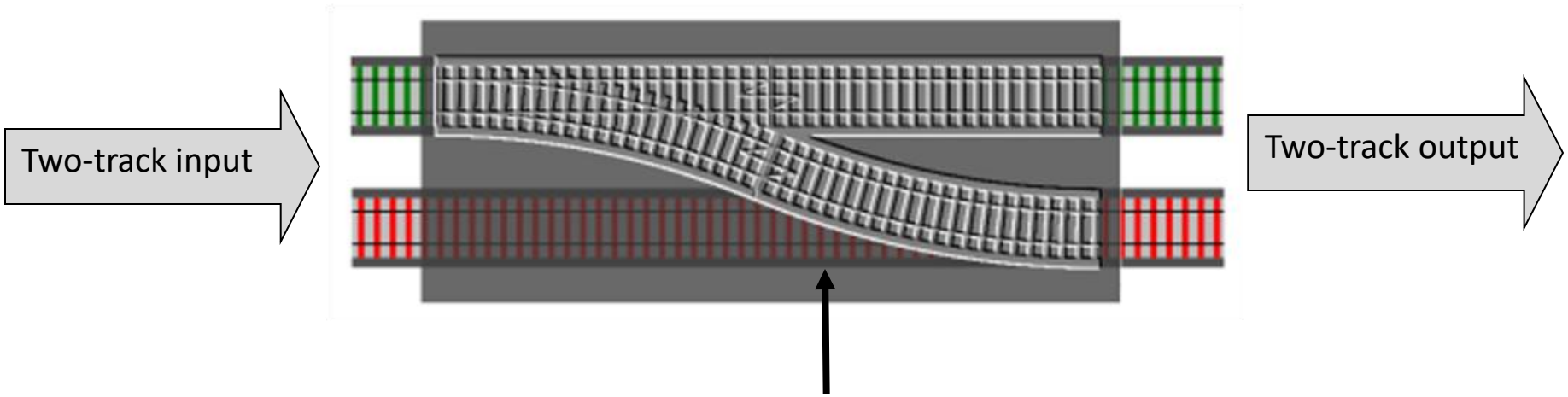
“Bind” is the answer!
Bind all the things!

FP’ers get excited by bind

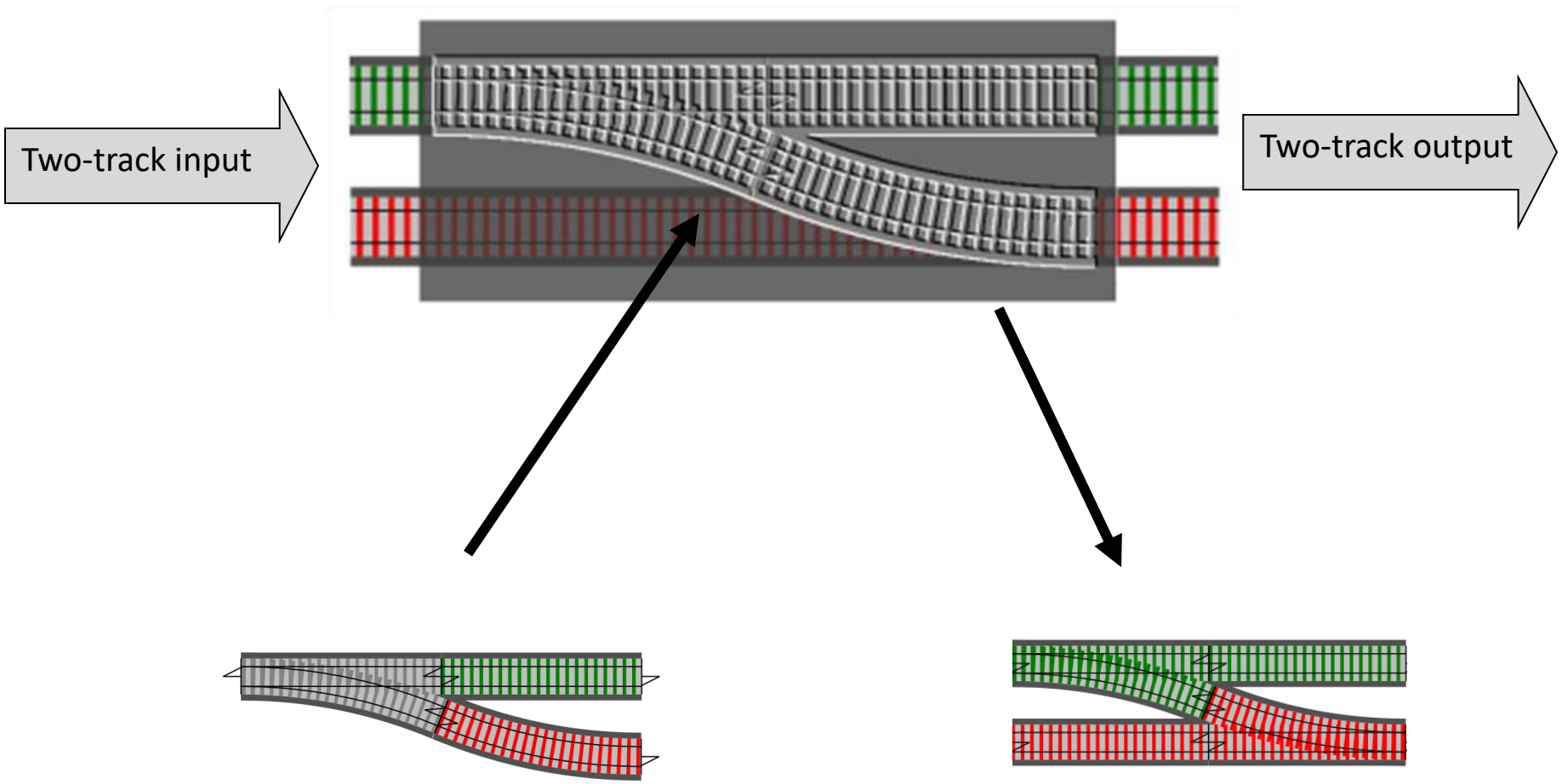


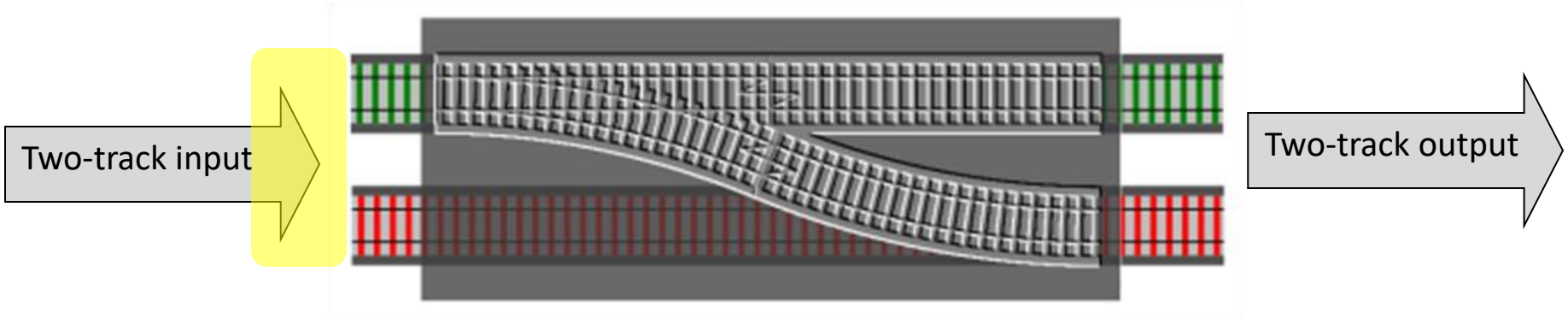


So how can we convert from the "before" case to the "after" case?

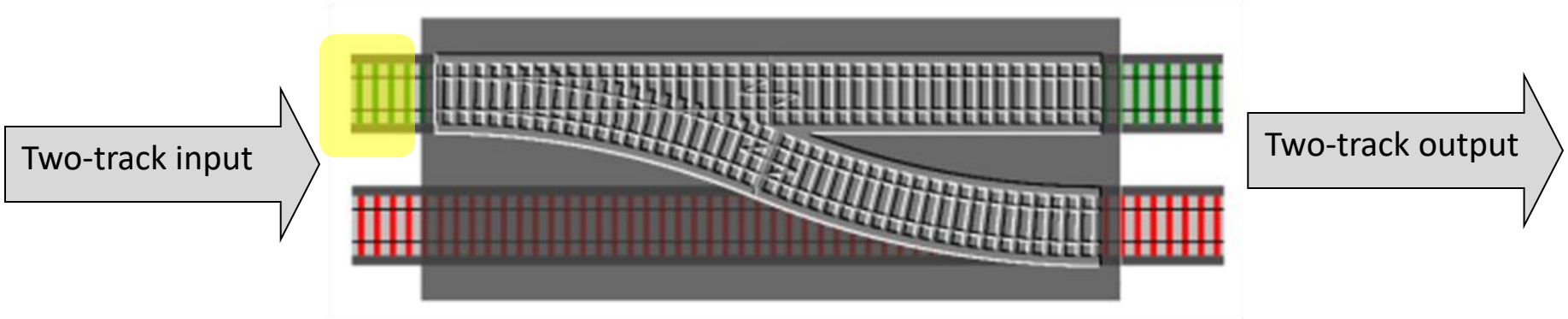


Slot for switch function

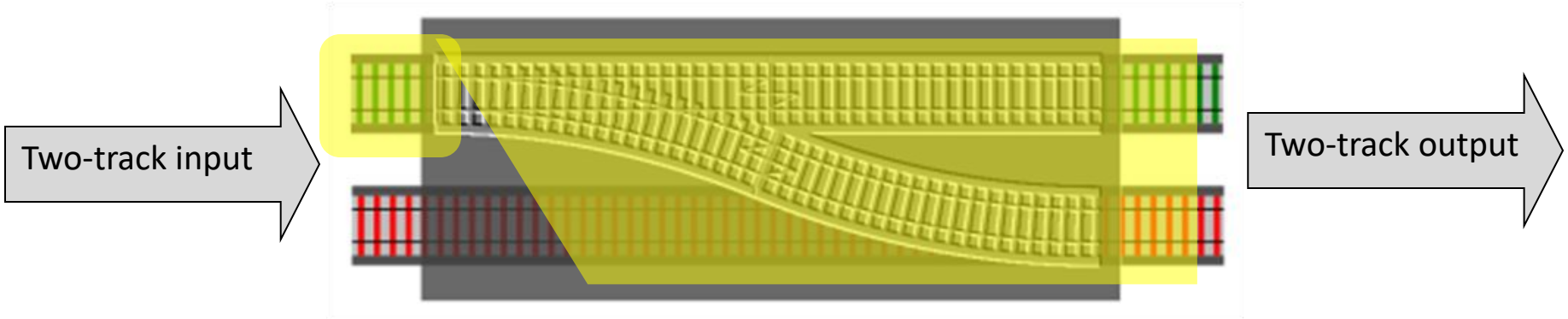

Two-track input
Two-track output



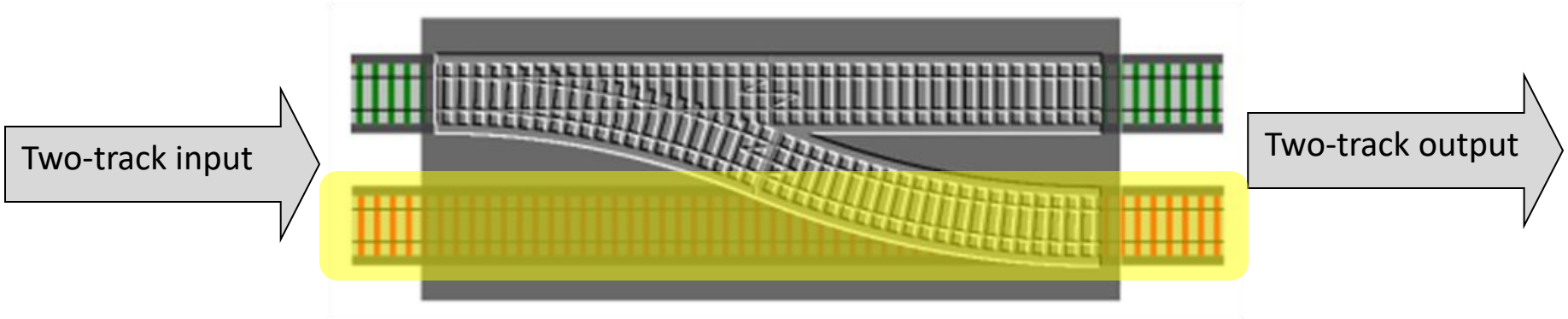

```
let bind nextFunction result =
   match result with
   | Ok s -> nextFunction s
   | Error e -> Error e
```



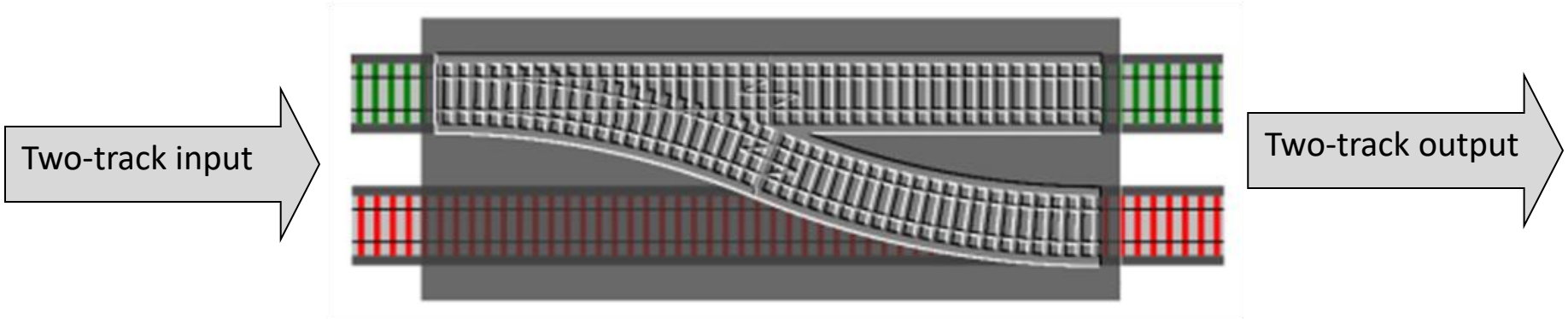

```
let bind nextFunction result =
   match result with
   | Ok s -> nextFunction s
   | Error e -> Error e
```




```
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
```



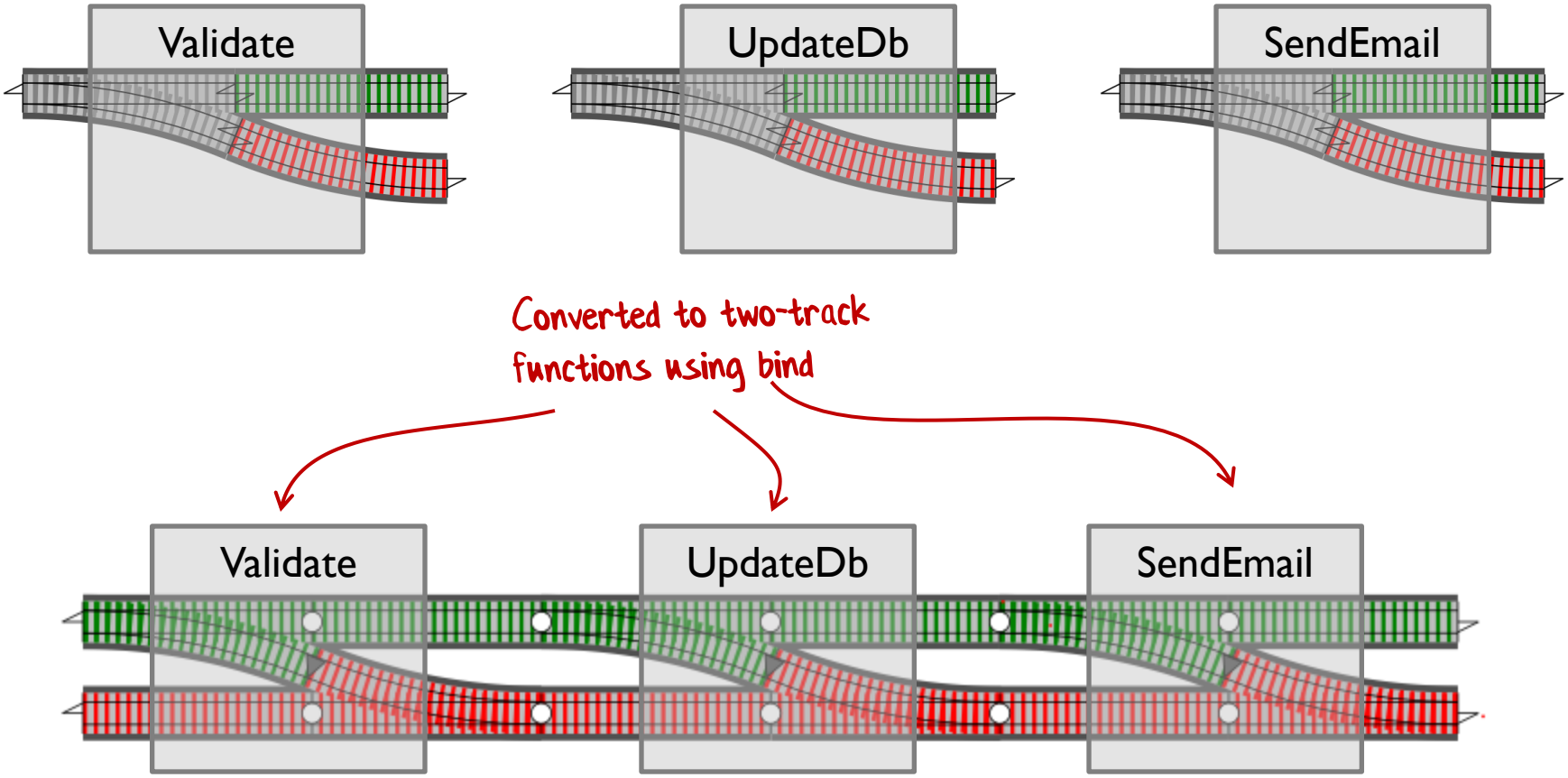

```
Result.bind : ('a -> Result<'b>) -> Result<'a> -> Result<'b>
```

Switch function

2-track input

2-track output

Composing switches - review



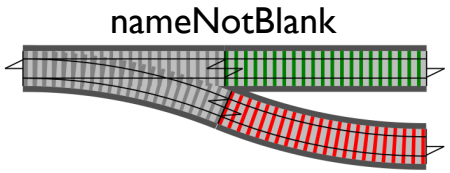
Bind example

Validating input

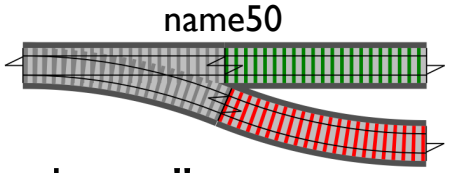
```
type Request = {
    Name : string
    Email : string
}
```

Is this data valid?

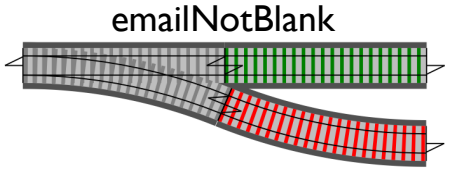
```
let checkNameNotBlank input =
  if input.Name = "" then
     Error "Name must not be blank"
  else Ok input
```

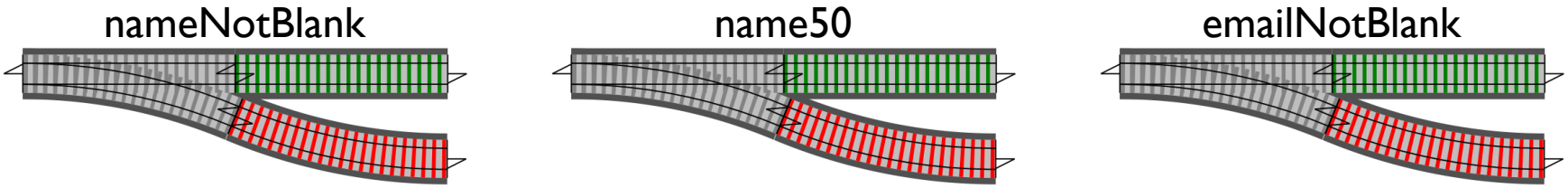



```
let checkName50 input =
  if input.Name.Length > 50 then
     Error "Name must not be longer than 50 chars"
  else Ok input
```

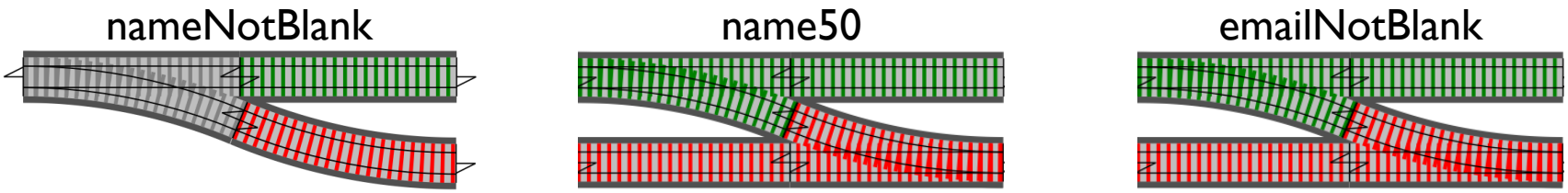



```
let checkEmailNotBlank input =
  if input.Email = "" then
     Error "Email must not be blank"
  else Ok input
```



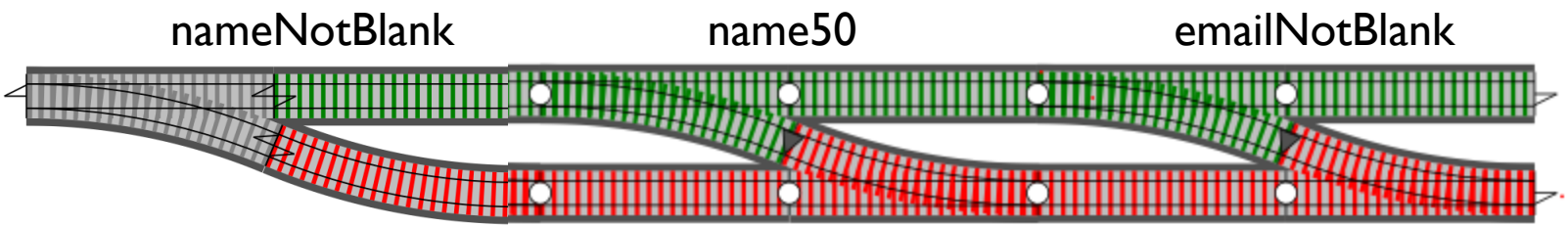



```
checkNameNotBlank (composed with)
            checkName50 (composed with)
                                  checkEmailNotBlank
```



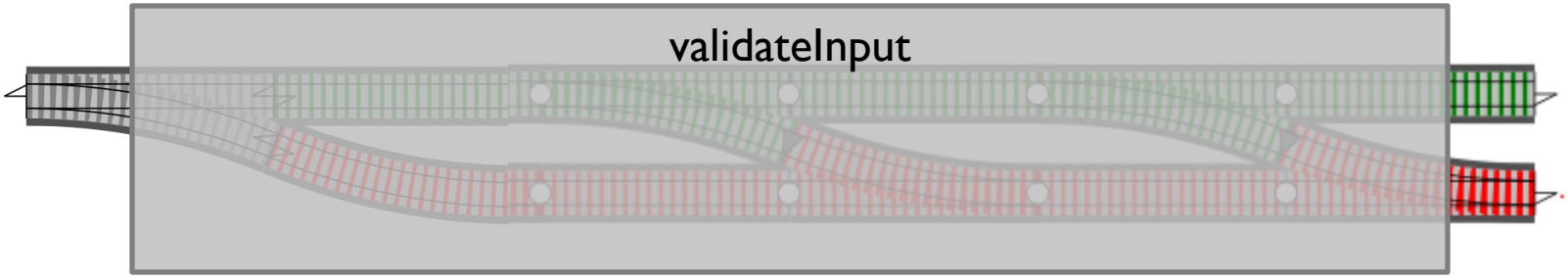

```
checkNameNotBlank

                    bind checkName50

                         bind checkEmailNotBlank
```

use "bind" to
convert to 2-track




```
request
|> checkNameNotBlank
|> Result.bind checkName50
|> Result.bind checkEmailNotBlank
```

then compose together

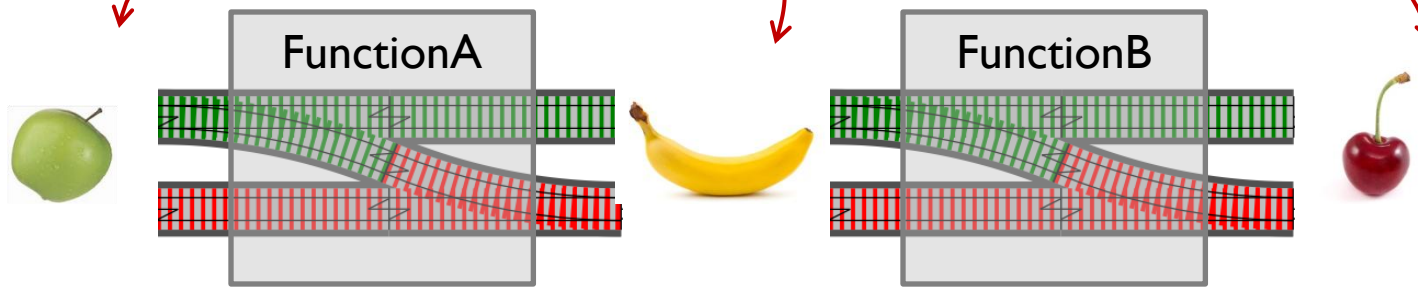


Define a function

```
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.

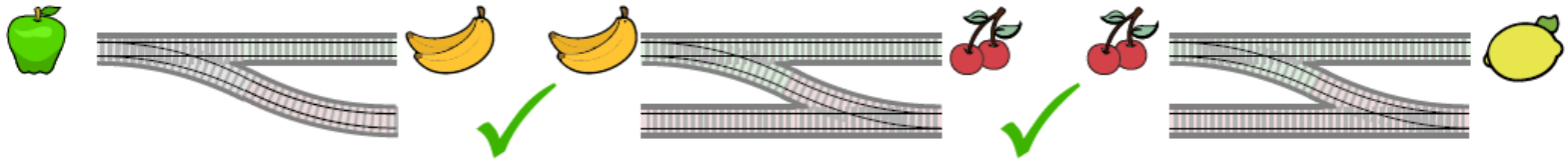


TEntity = Apple

TEntity = Banana

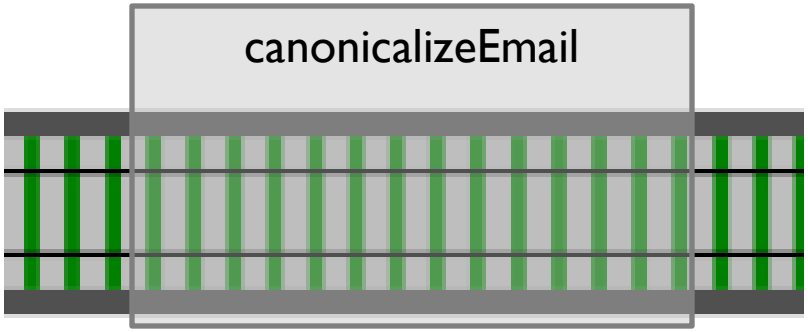
TEntity = Cherry

```
Result<'TEntity,string>
```



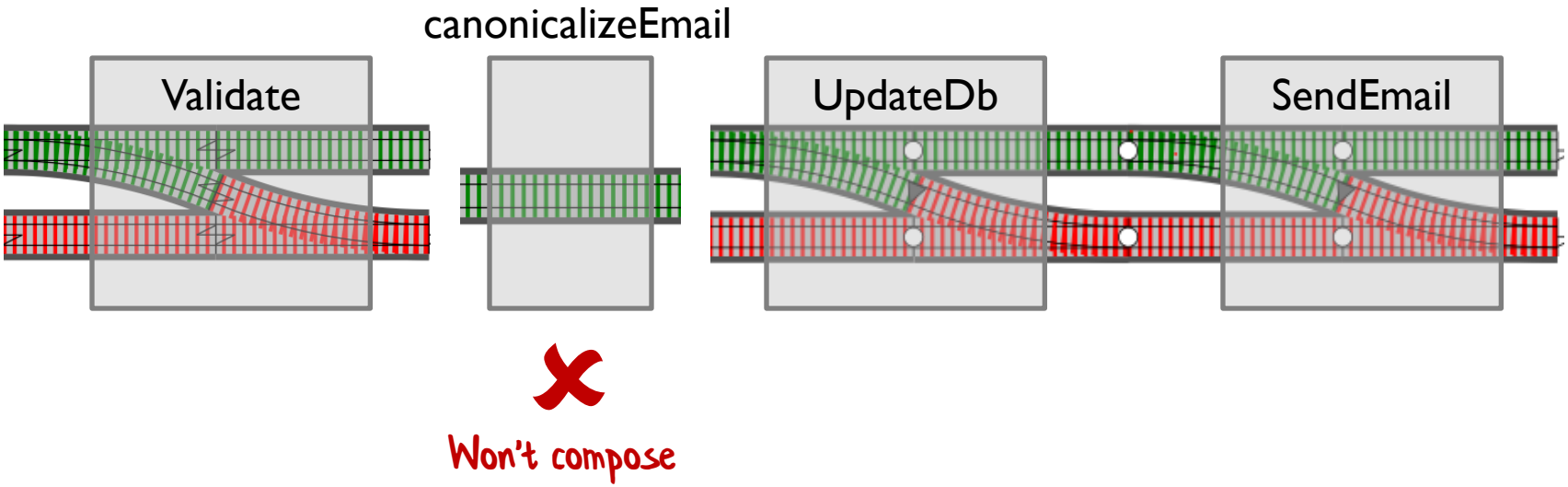
Exercise:
`02a-Exercise-ValidateRequest.fsx`

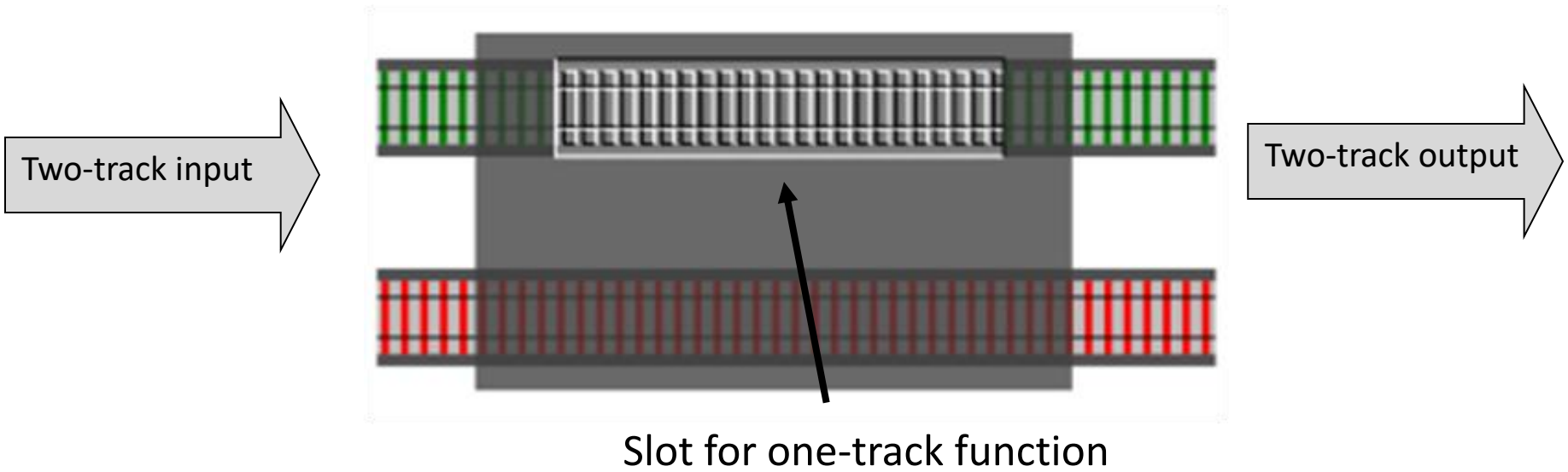
Mapping the success track

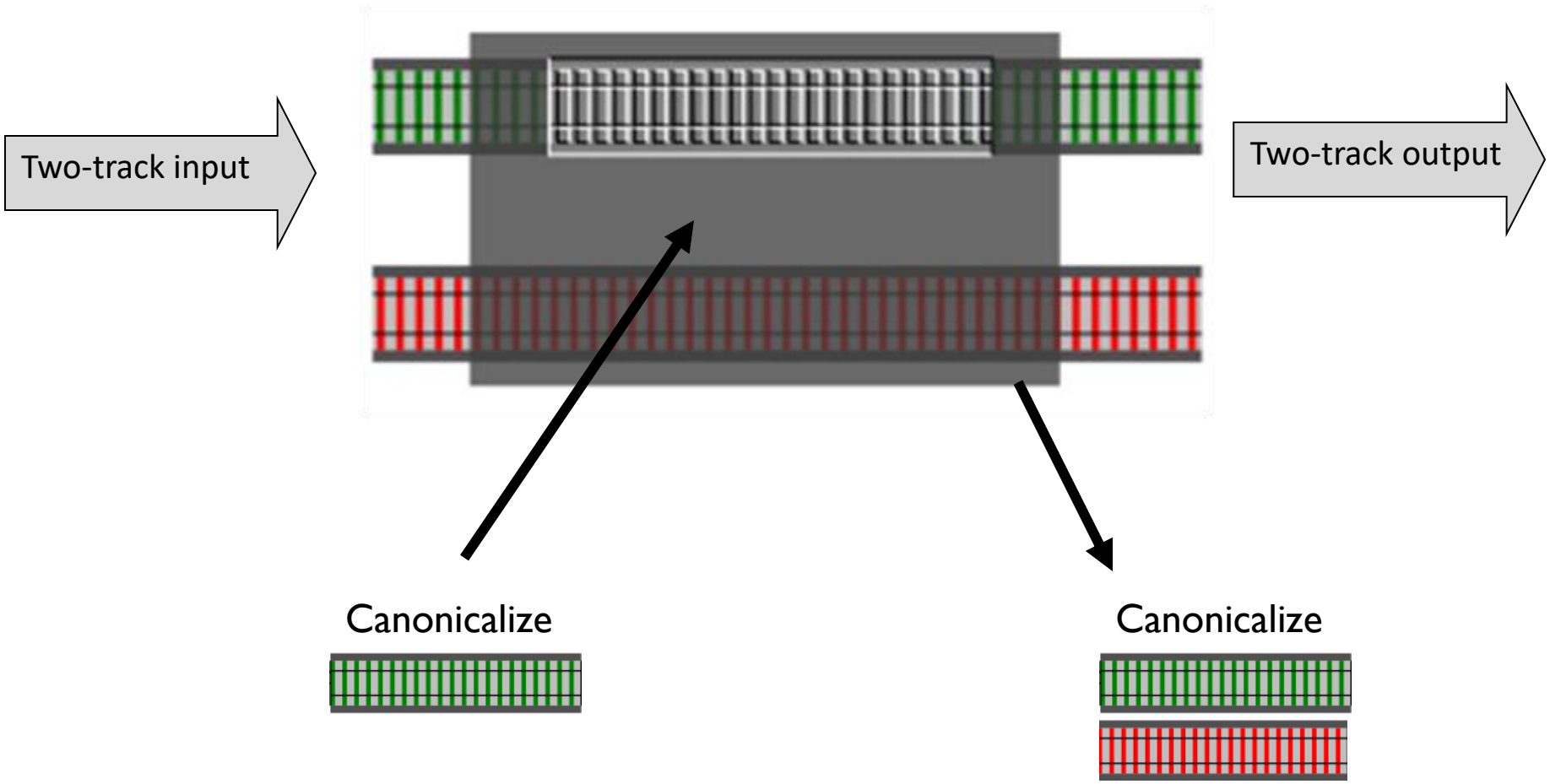


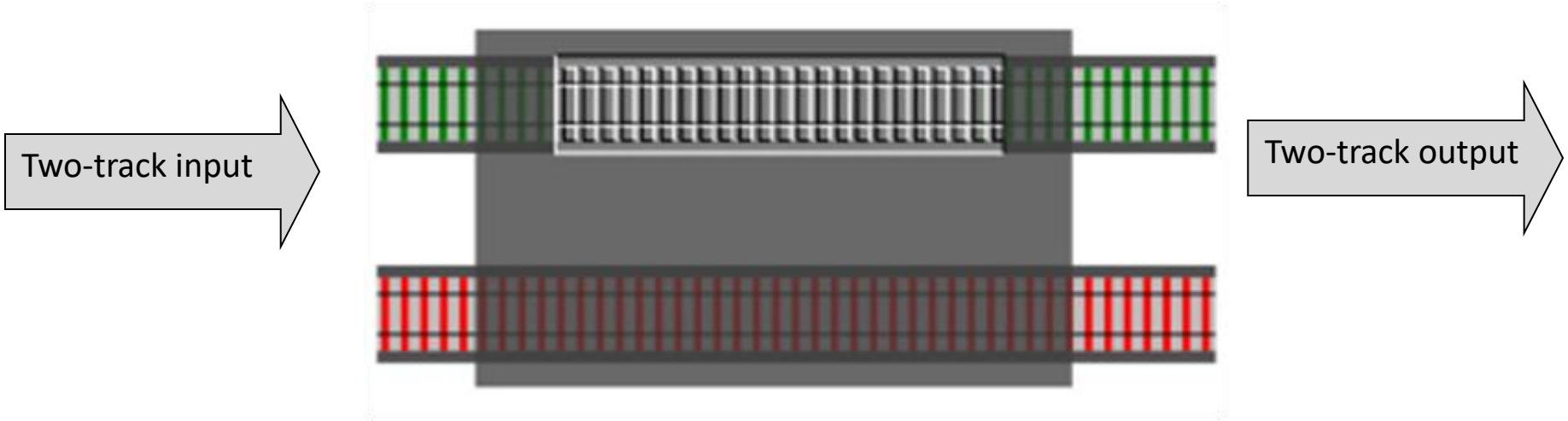

```
// trim spaces and lowercase
let canonicalizeEmail input =
   { input with email = input.email.Trim().ToLower() }
```

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

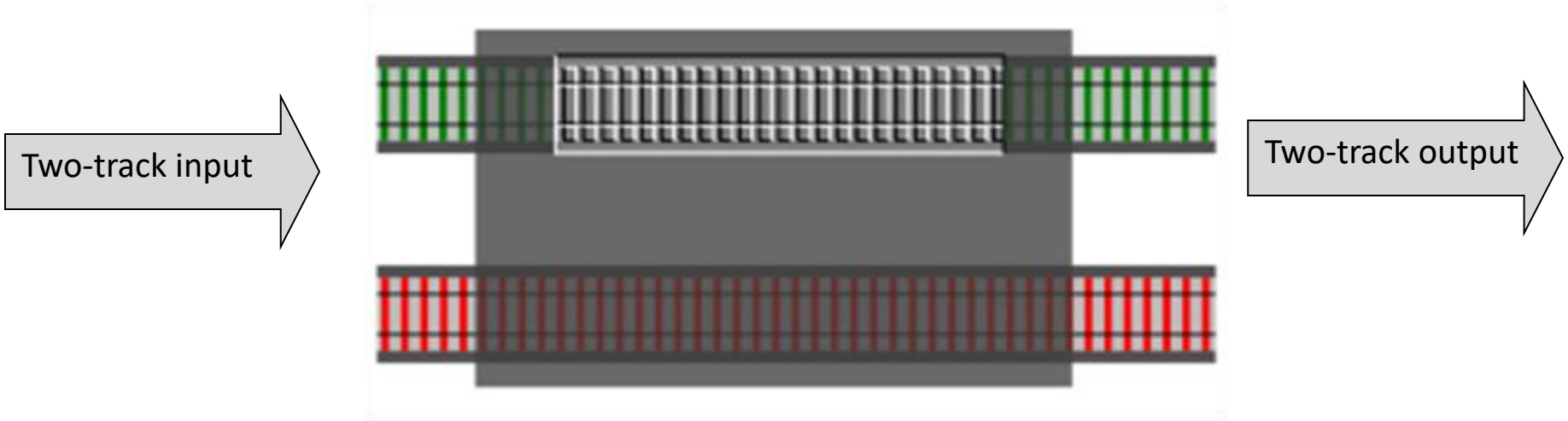

canonicalizeEmail
Validate
UpdateDb
SendEmail
Won't compose


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


Two-track input
Two-track output
Canonicalize
Canonicalize




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



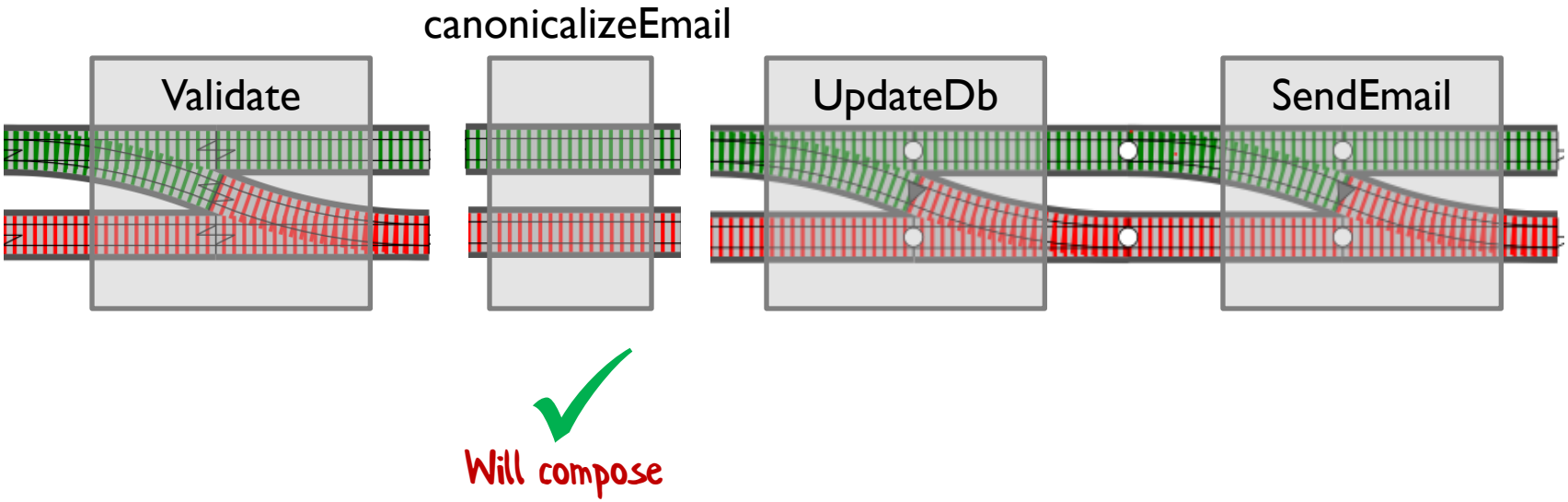

```
Result.map : ('a -> 'b) -> Result<'a,'c> -> Result<'b,'c>
```

Single track function

2-track input

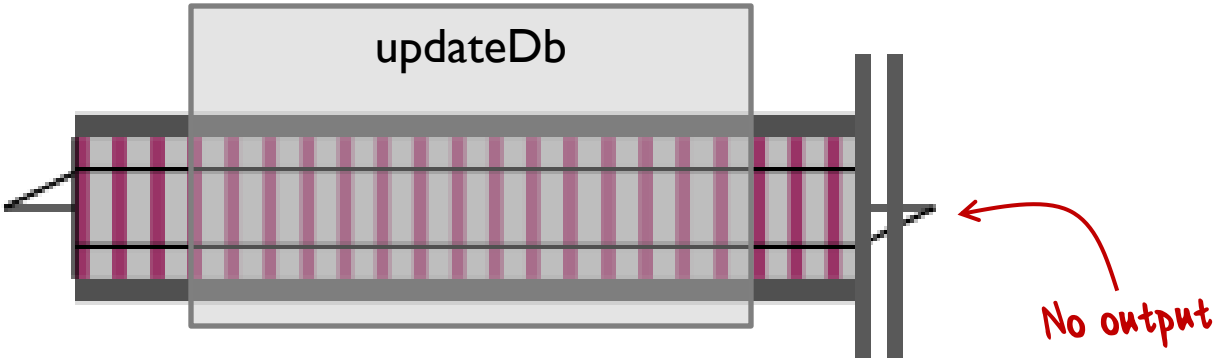
2-track output

Converting one-track functions



Converting dead-end functions

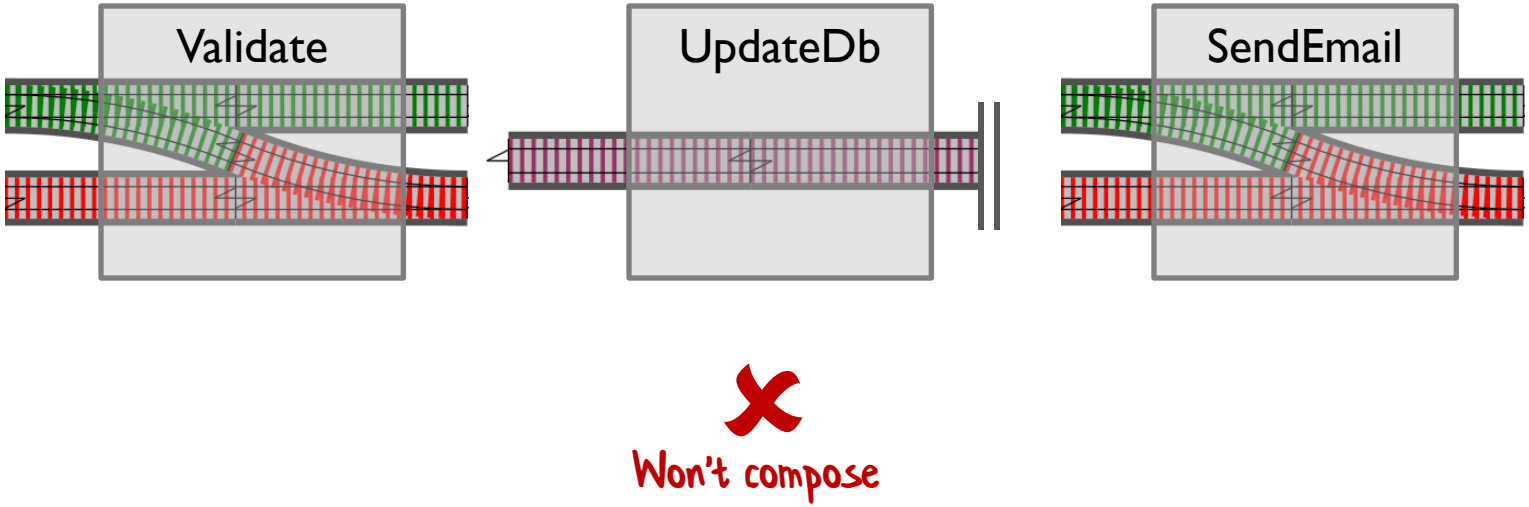
Converting dead-end functions



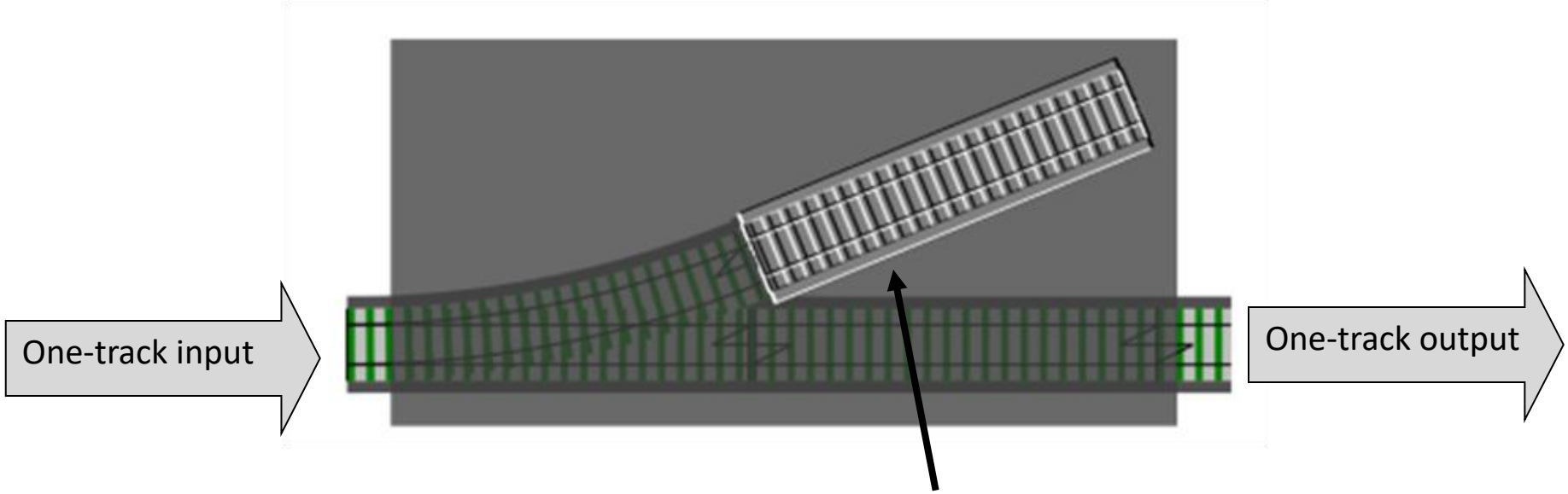

```
let updateDb request =
    // do something
    // return nothing at all
```

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

Converting dead-end functions

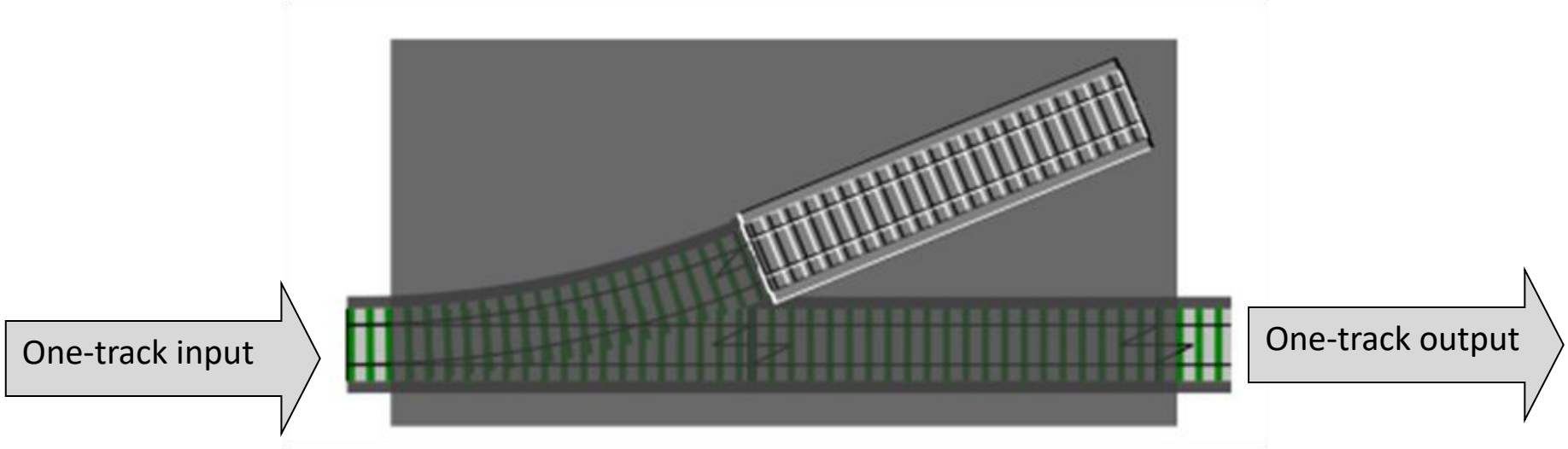


Converting dead-end functions



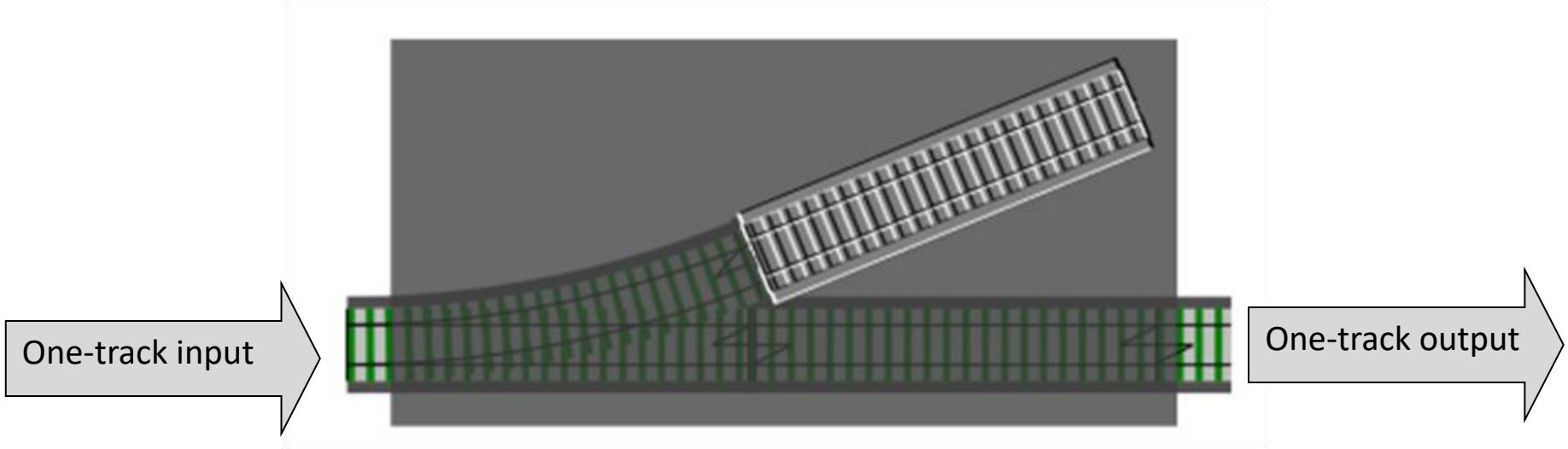
Slot for dead end function

Converting dead-end functions




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

Converting dead-end functions



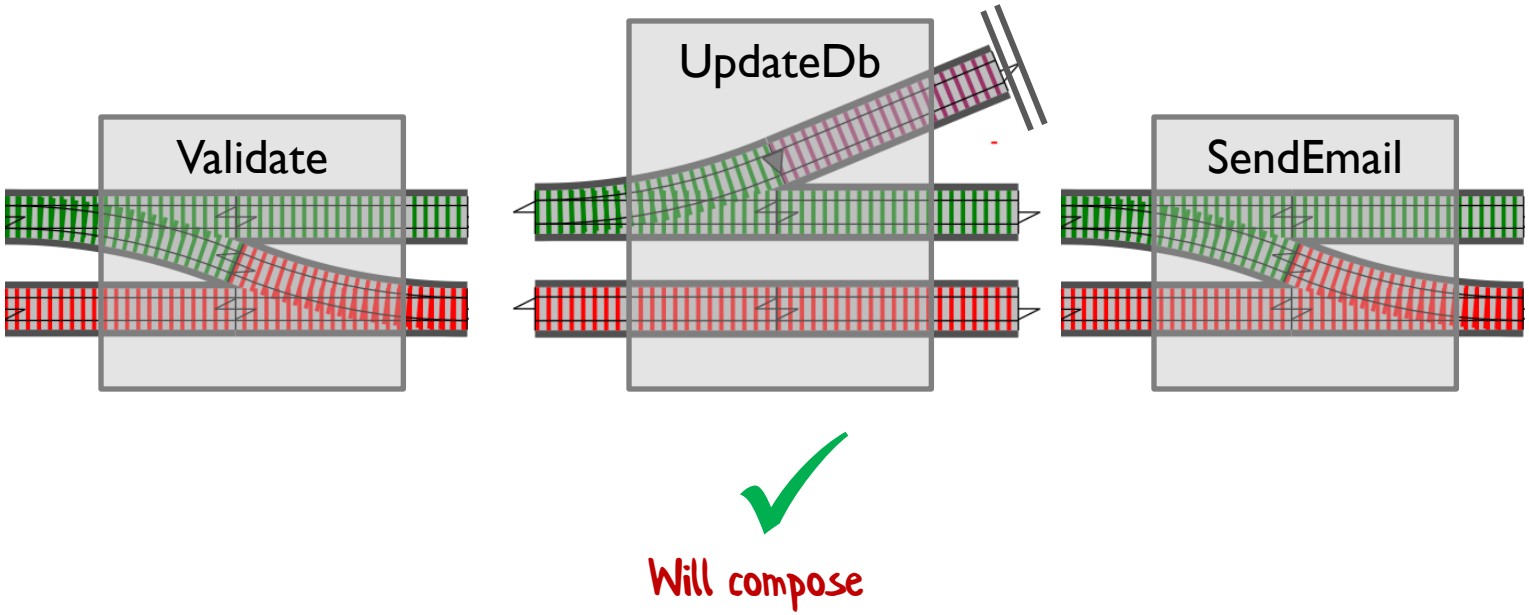

```
tee : ('a -> unit) -> 'a -> 'a
```

Dead end function

one-track input

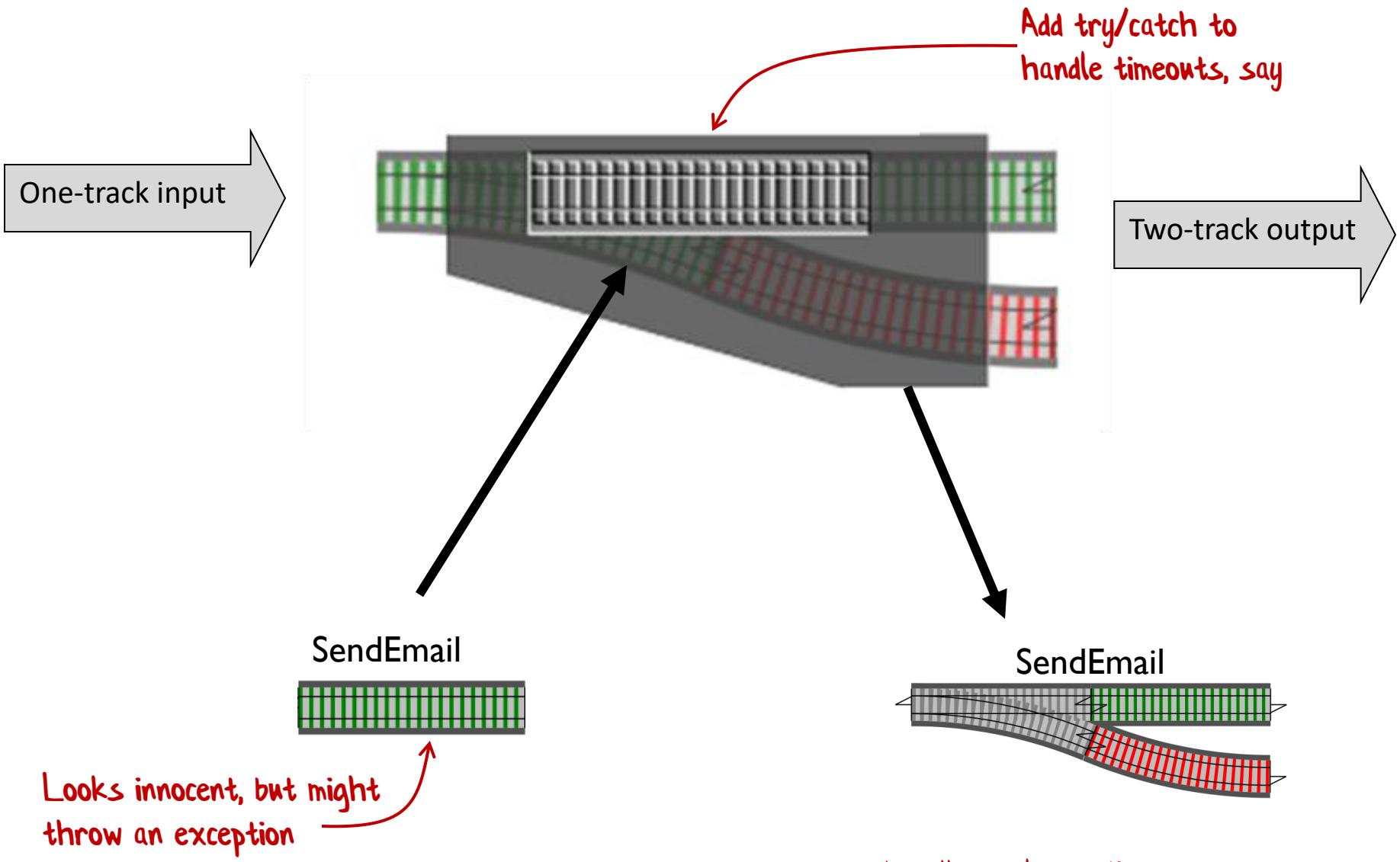
one-track output

Converting dead-end functions



Functions that throw exceptions

Functions that throw exceptions



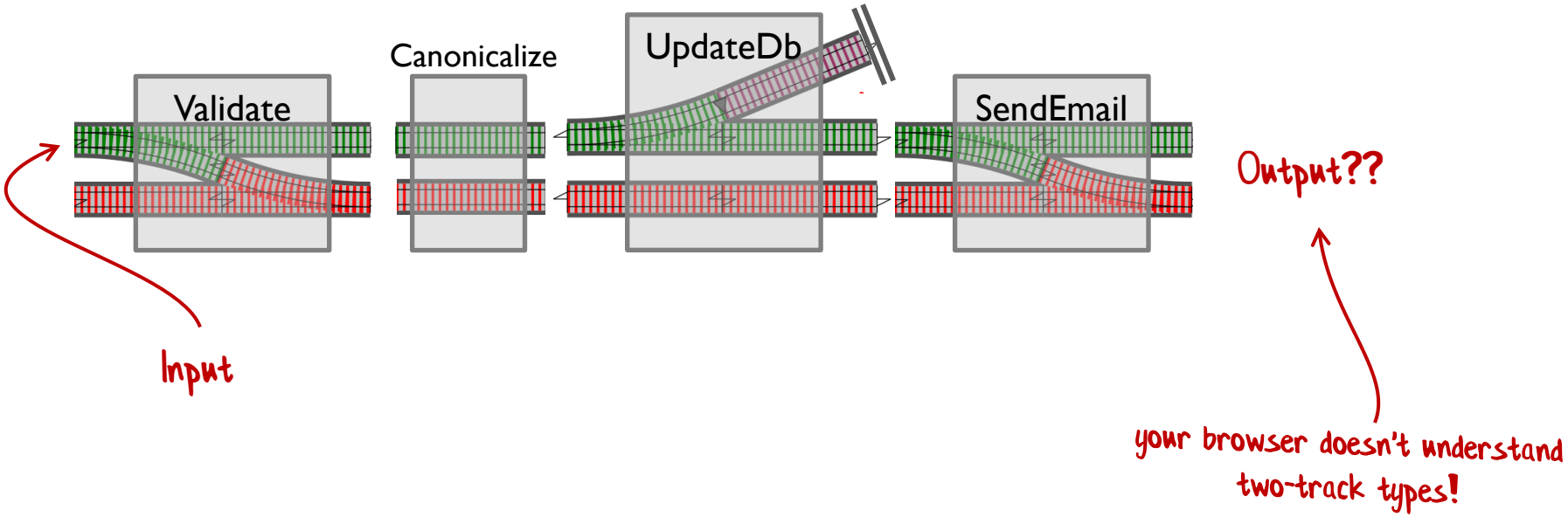
Guideline: Convert exceptions into Failures

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

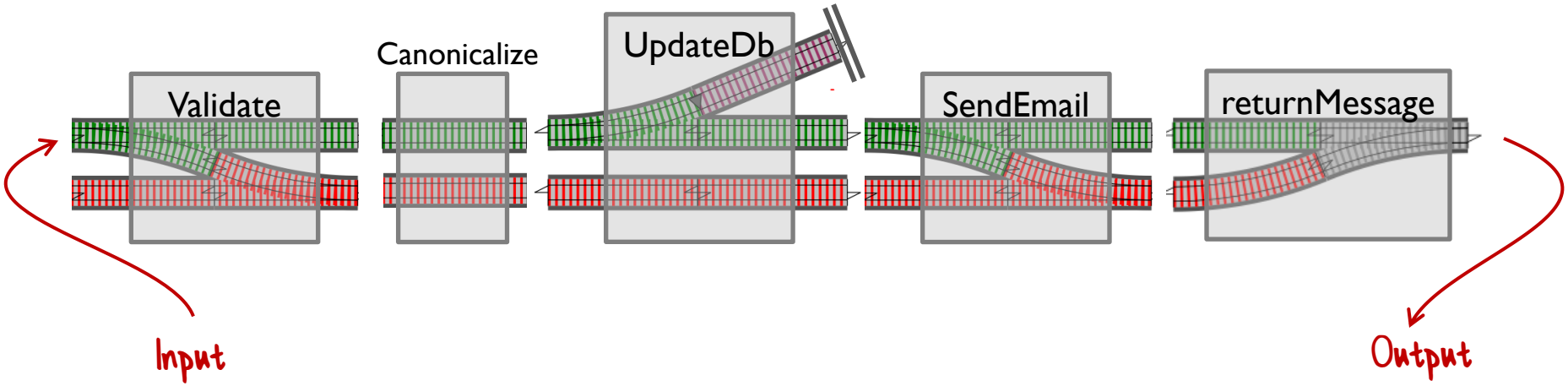
"Do or do not, there is no try".

Putting it all together

Putting it all together



Putting it all together




```
let returnMessage result =
  match result with
  | Ok obj -> OK obj.ToJson()
  | Error msg -> BadRequest msg
```

Demo:

`03a-RopWithStringError.fsx`

Exercise:

`03b-Exercise-RopWithErrorType.fsx`

Mapping the error track

Converting to a common error type

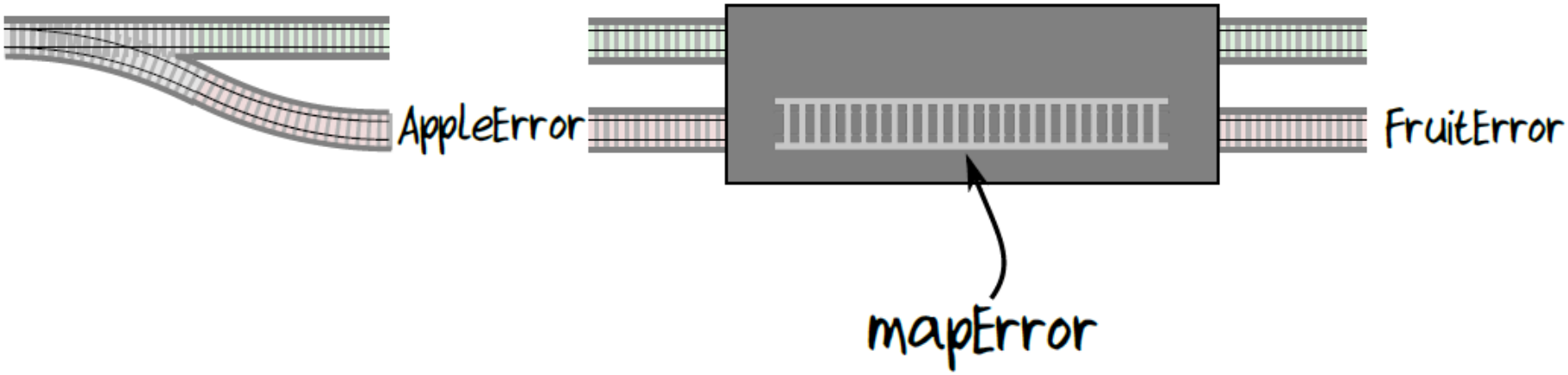
Make sure all errors are the same type

- The error track has the same type all the way along the track.
- So, we may need to change the error types to make them compatible.
- This what "`Result.mapError`" is for

```
type FunctionA =
   Apple -> Result<Bananas,AppleError>

type FunctionB =
   Bananas -> Result<Cherries,BananaError>
```

```
// define a common superset
type FruitError =
| AppleErrorCase of AppleError
| BananaErrorCase of BananaError
```

```
type FunctionA =
   Apple -> Result<Bananas,FruitError>

type FunctionB =
   Bananas -> Result<Cherries,FruitError>
```


AppleError
FruitError
mapError

```
let functionA' input =
  input
  |> functionA // original function
  |> Result.mapError AppleErrorCase
  // Apple -> Result<Bananas,FruitError>

// do the same for the other function
let functionB' input =
  input
  |> functionB // original function
  |> Result.mapError BananaErrorCase
  // Bananas -> Result<Cherries,FruitError>

// now they can be composed!
```

Review of "implementing workflows"

Demo:

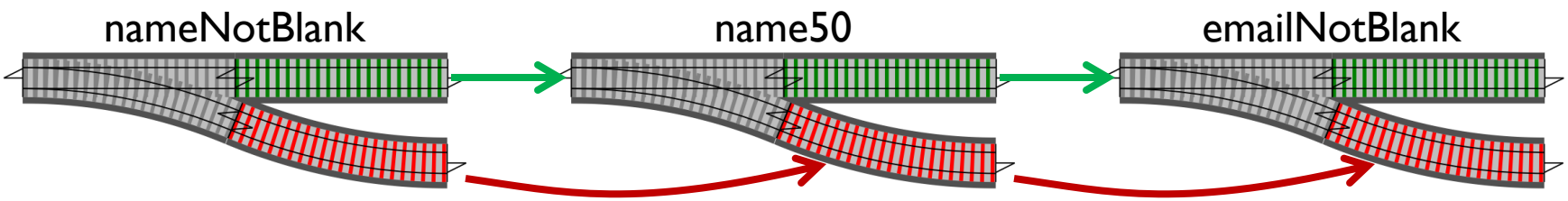
`04a-RopWithTicTacToe.fsx`

Exercise:

`04b-Exercise-RopWithCoffeeMaker.fsx`

Validation

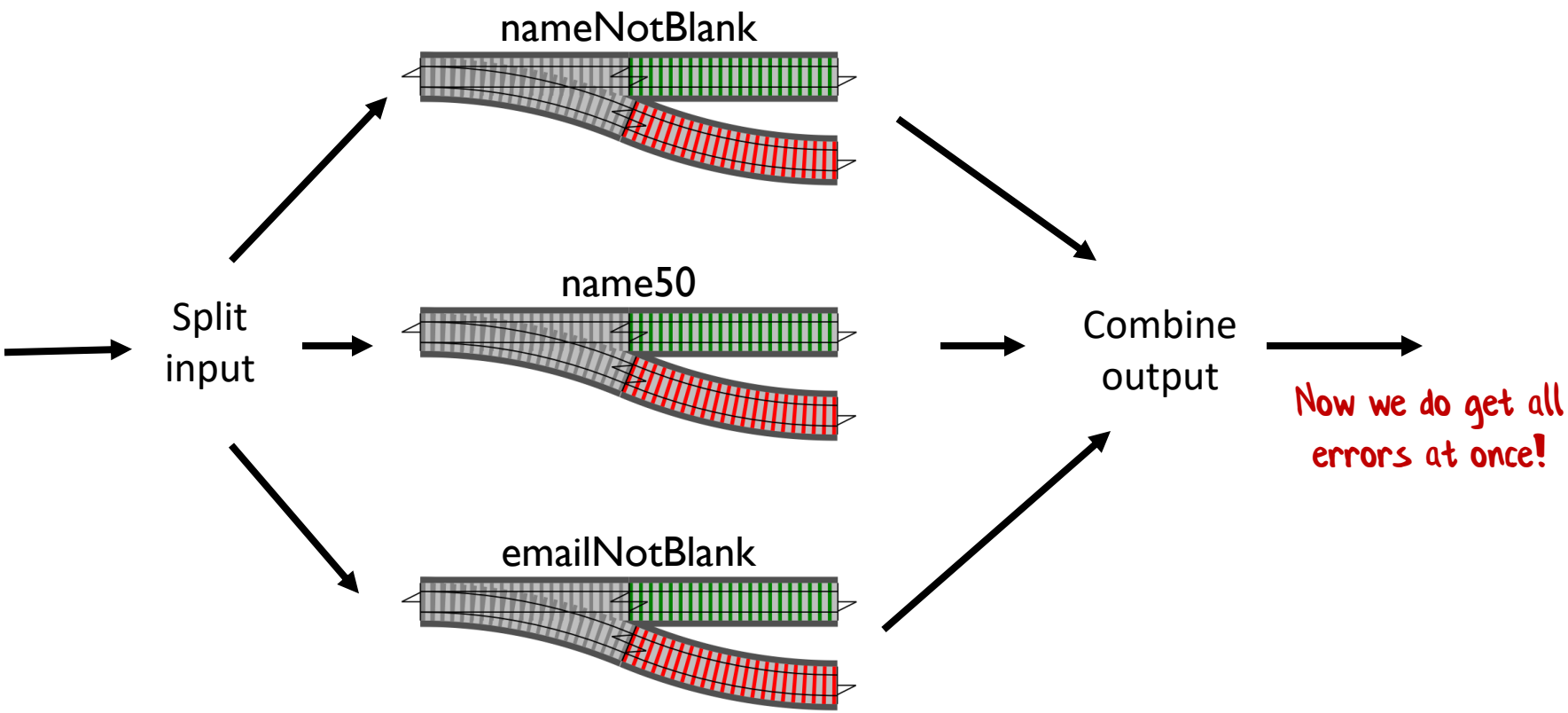
Parallel validation



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



Now we do get all errors at once!

... But how to combine?

How does it work?

- A bunch of functions that return a ValidationType
- A constructor
- Use <!> and <*>

How does it work?

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

let firstR = validateFirst first
   // returns Validation<First,_>
let lastR = validateLast last
   // returns Validation<Last,_>
let ageR = validateAge age
   // returns Validation<Age,_>
let personR =
   createPerson <!> firstR <*> lastR <*> ageR
   // returns Validation<Person,_>
```

Demo:

`05a-ValidationWithMultipleErrors.fsx`

Exercise:

`05b-Exercise-DtoValidation.fsx`

Computation expressions

Demo:

`06a-ResultComputationExpression.fsx`

Different kinds of errors

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

Kinds of error: Domain errors

- **Domain errors** are to be expected as part of the business process
- Must be included in the design of the domain, just like anything else.
- The business will already have procedures in place to deal with this kind of error, and so the code will need to reflect these processes.
- Diagnostics/Stack trace are not needed

Kinds of error: Panics

- **Panics** leave the system in an unknown state
  - System errors (e.g. "out of memory") or programmer oversight (e.g. "divide by zero," "null reference").
- Handled by abandoning the workflow and raising an exception which is then caught and logged at the highest appropriate level.
- Diagnostics/Stack trace are needed

Kinds of error: Infrastructure error

- **Infrastructure errors** are expected as part of the architecture
- Not part of any business process and are not included in the domain.
  - Network timeout, authentication failure, etc.
- Sometimes modeled as part of the domain, and sometimes treated as panics. If in doubt, ask a domain expert!

Summary:
Be aware of different kinds of errors

Don't use Result for everything!

See "Against Railway-Oriented Programming"