Error Processing: An Exercise in Functional Design

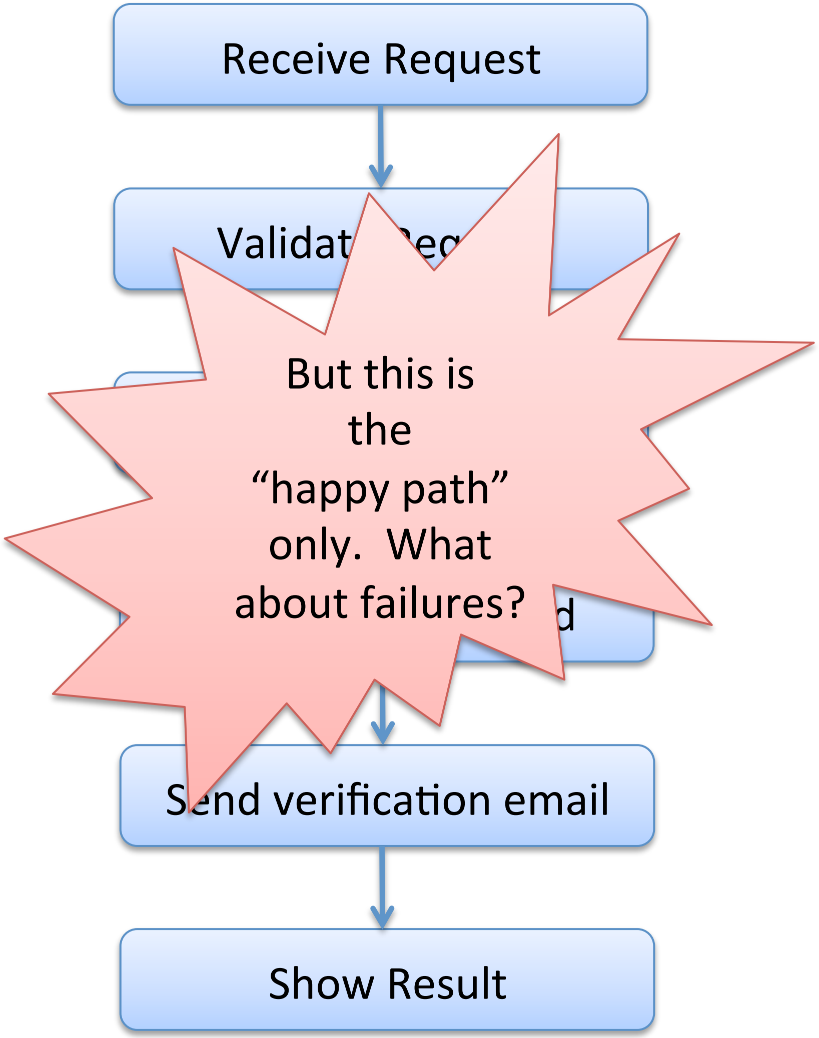
COS 326
David Walker

This lecture from a great blog on F#:
http://fsharpforfunandprofit.com/posts/recipe-part1/

The Task

- Imagine you are designing a front end for a database that takes update requests.
  - A user submits some data (userid, name, email)
  - Check for validity of name, email
  - Update user record in database
  - If email has changed, send verification email
  - Display end result to user

In Pictures



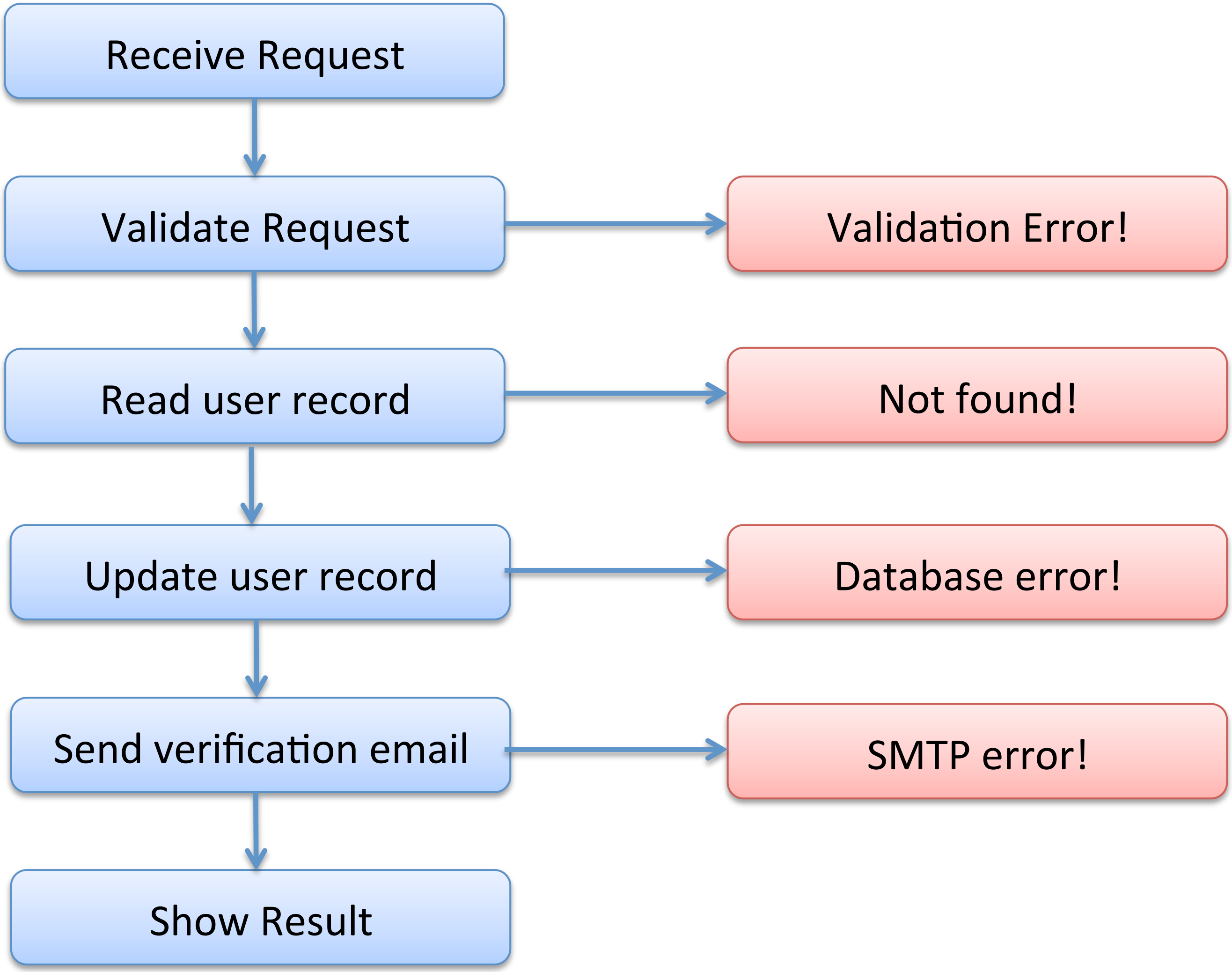
Receive Request

Send verification email

Show Result

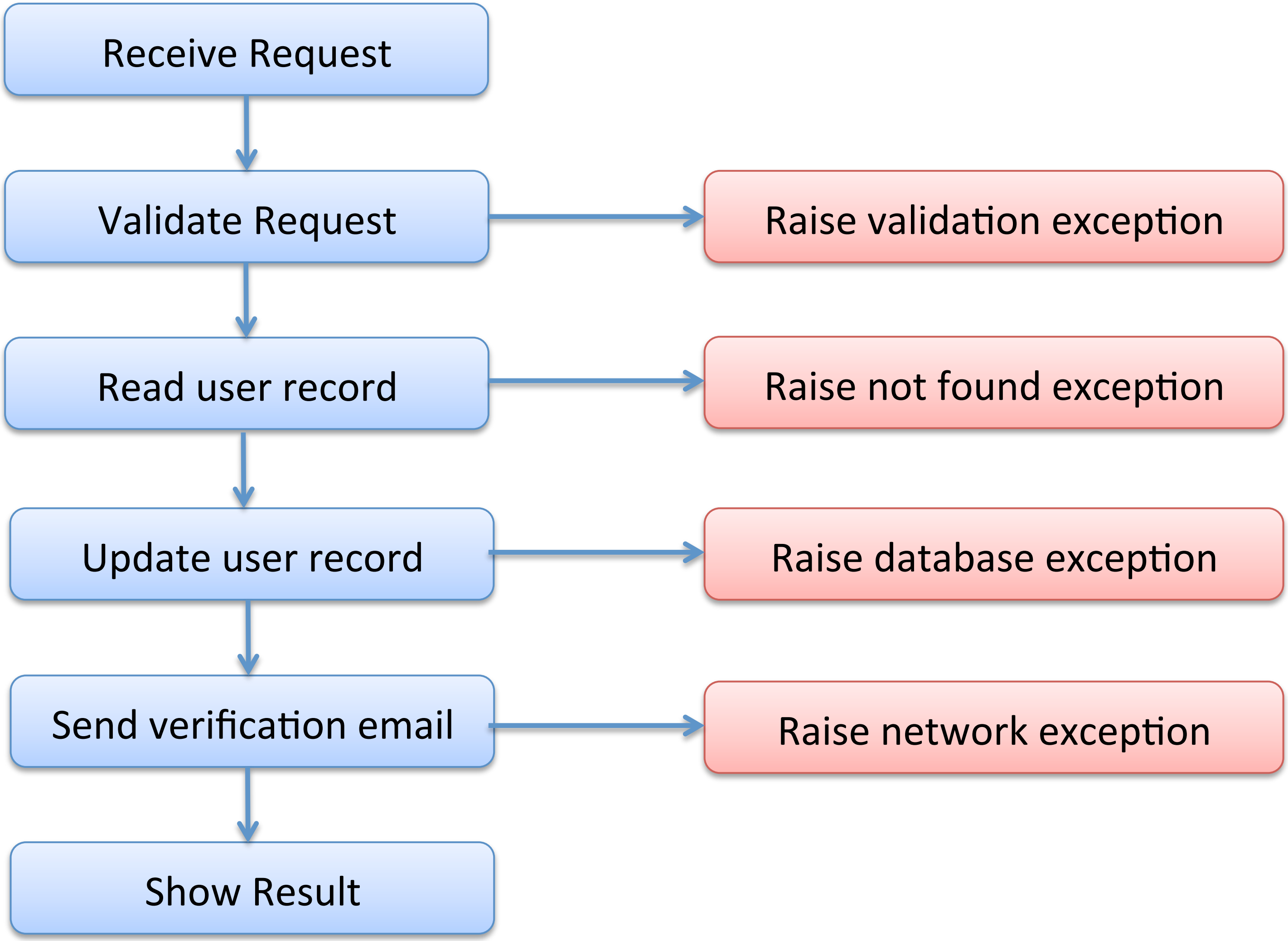
But this is the “happy path” only. What about failures?

In Pictures



Receive Request
Validate Request
Validation Error!
Read user record
Not found!
Update user record
Database error!
Send verification email
SMTP error!
Show Result

One solution



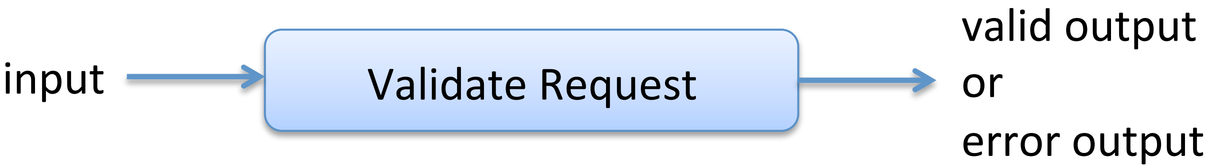
The trouble with exceptions

People forget to catch them!

- applications fail
- sadness ensues
- See "A type-based analysis of uncaught exceptions" by Pessaux and Leroy.
  - Uncaught exceptions: a big problem in OCaml (and Java!)

In a more functional approach, the full behavior of a program is determined exclusively *by the value it returns*, not by its "effect"

Functional Error Processing

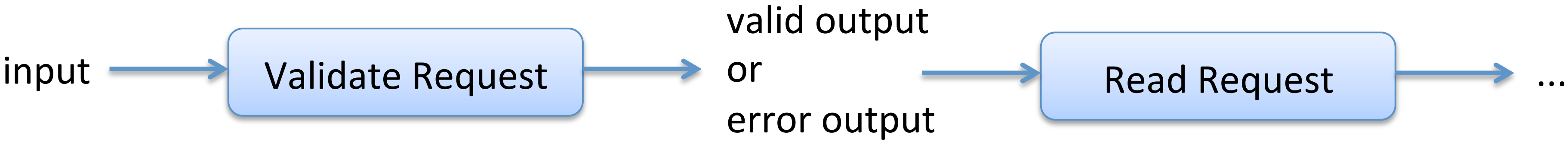


input

Validate Request

valid output
or
error output

The Challenge: Composition



input

Validate Request

valid output
or
error output

Read Request

...

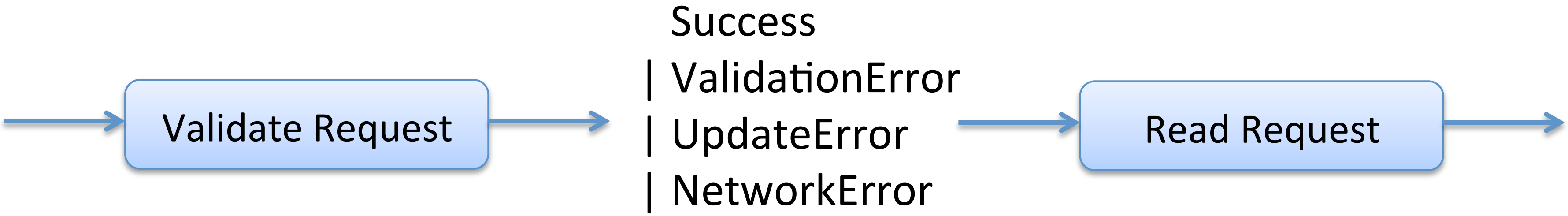
One Possibility

Define a datatype to represent all outputs:

```
type result =
  Success | ValidationError | UpdateError | NetworkError
```

But:

- not very reuseable (very specific set of errors)
- adding a new error is irritating
- every function in the chain must process all possible errors as inputs:



Validate Request

Success
| ValidationError
| UpdateError
| NetworkError

Read Request

A better idea:
Generic errors & error-processing library

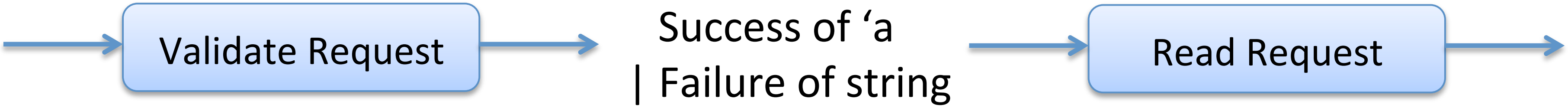
A generic result type:

```
type (‘a, ‘b) result =
      Success of  ‘a
    | Failure of ‘b
```

Specialized to string errors:

```
type ‘a eresult = (‘a, string) result
```

A processing pipeline:



Validate Request

Success of ‘a
| Failure of string

Read Request

An Example Pipeline Function

```
type (‘a, ‘b) result = Success of  ‘a | Failure of ‘b
type ‘a eresult = (‘a, string) result

type request = {name:string; email:string}

let validate input =
  if input.name = “” then
      Failure “name must not be blank”
  else if input.email = “” then
      Failure “email must not be blank”
  else
      Success input
```

```
validate : request -> request eresult
```

Note: we really don’t want to have match on a possibly erroneous input every single time, so we assume a good input gets passed in, a possibly erroneous result returned

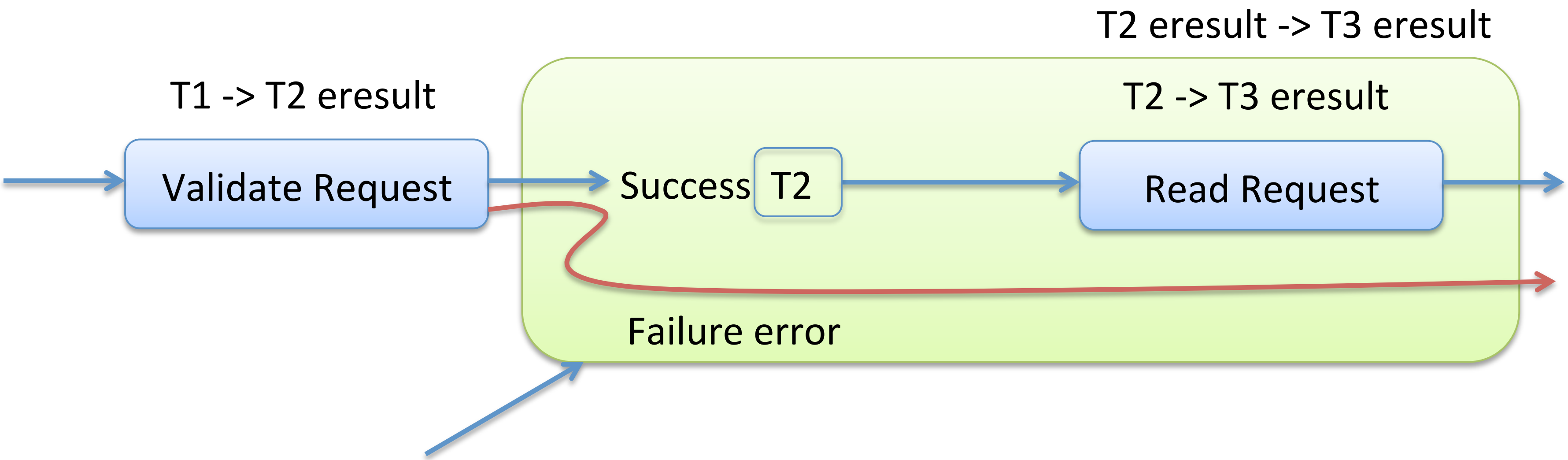
An Example Pipeline Function

```
type (‘a, ‘b) result = Success of  ‘a | Failure of ‘b
type ‘a eresult = (‘a, string) result

type request = {name:string; email:string}

let validate input =
  if input.name = “” then
      Failure “name must not be blank”
  else if input.email = “” then
      Failure “email must not be blank”
  else
      Success input
```

```
validate : request -> request eresult
```

```
in general,     f : T1 -> T2 eresult
```

Composition



Goal: Create a bypass combinator to convert an ‘a -> ‘b eresult function in to a function with type ‘a eresult -> ‘b eresult

```
let bind f =
   fun result ->
      match result with
            Success v -> f v
          | Failure s -> result
```

```
bind :  (‘a -> ‘b eresult) -> (‘a eresult -> ‘b eresult)
```

```
let (>>=) x f = bind f x
```

similar to |>

```
>>= :  ‘a eresult -> (‘a -> ‘b eresult) -> ‘b eresult
```

Using the bypass combinator

```
let validate_name1 input =
  if input.name = “” then Failure “no name”
  else Success input

let validate_name2 input =
  if String.length (input.name) > 50 then Failure “name too long”
  else Success input

let validate_email input =
  if input.email = “” then Failure “no email”
  else Success input
```

```
let validator input =
    input    |> validate_name1
             >>= validate_name2
             >>= validate_email
```

```
validator :  request -> request eresult
```

An Alternative

```
let (>=>) f1 f2 =
  fun x ->
      match f1 x with
         Success s -> f2 s
       | Failure f -> Failure f
```

```
>=> : (‘a eresult -> ‘b eresult) -> (‘b eresult -> ‘c eresult) -> (‘a eresult -> ‘c eresult)
```

similar to ordinary
function composition,
but for eresults

```
let validator =
        validate_name1
  >=> validate_name2
  >=> validate_email
```

```
validator :  request -> request eresult
```

An Error-Processing Library

```
type (‘a, ‘b) result = Success of ‘a | Failure of ‘b

type ‘a eresult = (‘a, string) result

(|>) : ‘a -> (‘a -> ‘b) -> ‘b

bind :   (‘a -> ‘b eresult) -> (‘a eresult -> ‘b eresult)

(>>=) :  ‘a eresult –> (‘a -> ‘b eresult) -> ‘b eresult

(>=>) : (‘a eresult -> ‘b eresult) -> (‘b eresult -> ‘c eresult) -> (‘a eresult -> ‘c eresult)
```

```
return : ‘a -> ‘a eresult                           (* successful with ‘a *)

fail : string -> ‘a eresult                         (* automatic failure *)

map : (‘a -> ‘b) -> (‘a eresult -> ‘b eresult)      (* convert an error-free function *)

(>>) : (‘a -> ‘b) -> (‘b -> ‘c) -> (‘a -> ‘c)       (* composition *)
```

A coincidence?

error computations:	map : (‘a -> ‘b) -> ‘a eresult -> ‘b eresult
list computations:	map : (‘a -> ‘b) -> ‘a list -> ‘b list

error computations:	bind : (‘a -> ‘b eresult) -> (‘a eresult -> ‘b eresult)
list computations:	bind : (‘a -> ‘b list) -> (‘a list -> ‘b list)

error computations:	return : ‘a -> ‘a eresult
list computations:	return : ‘a -> ‘a list

Monads

- A monad is a data type + functions bind and return that satisfies certain equational laws:

```
(return a >>= f)  == f a
```

```
m >> return == m
```

```
m >>= (fun x -> k x >>= h) == m >>= k >>= h
```

- In this lecture, we saw how a monad library helped us handle one kind of effect: an exception
- Monads are a general mechanism for handling effects
- Haskell has a built in syntax for monads and has structured their libraries so that a function with type a -> b has no effect. Only functions with type a -> M b for certain monads M have effects

Summary

Functi

SCORE: OCAML 4, JAVA 0

|> : ‘a -> (‘

bind :   (‘a -> ‘b er a er -> ‘b eres lt)

>>= : ‘a eresult –> (‘a -> ‘b eresult) -> ‘b eresult

>=> : (‘a eresult -> ‘b eresult) -> (‘b eresult -> ‘c eresult) -> (‘a eresult -> ‘c eresult)