Error Processing:
An Exercise in Functional Design

COS 326
Andrew Appel

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

slides copyright 2013-2015 David Walker and Andrew W. Appel
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

Validate Request

Read user record

Update user record

Send verification email

Show Result

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

Receive Request

Validate Request

Read user record

Update user record

Send verification email

Show Result

Validation Error!

Not found!

Database error!

SMTP error!
One solution

1. Receive Request
2. Validate Request
   - Raise validation exception
3. Read user record
   - Raise not found exception
4. Update user record
   - Raise database exception
5. Send verification email
   - Raise network exception
6. Show Result
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!)
  • (not a big problem in C. Why not? 😞)

In a more functional approach, the full behavior of a program is determined exclusively *by the value it returns*, not by its “effect”
Explicitly return "good" result or error. If we use OCaml data types to represent the two possibilities we will force the client code to process the error (or get a warning from the OCaml type checker).
Functional Error Processing

Notice input and output aren’t the same type. On the surface, this makes it look awkward to compose a series of such steps, but:

*Good abstractions are compositional ones.*

Let’s design a generic library for error processing that is *highly reusable* and *compositional*.
Functional Error Processing

The Challenge: Composition
One Possibility

Define a datatype to represent all outputs:

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

But:

- not very reusable (very specific set of errors)
- adding a new error is irritating
- every function in the chain must process all possible errors as inputs:
A better idea:
Generic errors & error-processing library

A generic result type:

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

Specialized to string errors:

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

A processing pipeline:

1. Validate Request
2. Success of ‘a
   | Failure of string
3. Read Request
An Example Pipeline Function

```haskell
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
```

// result is a Success of 'a or Failure of string

```haskell
type 'a eresult = ('a, string) result
```

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

\[ T1 \rightarrow T2 \text{ eresult} \]

is the type of a possibly-erroneous function that takes a \( T1 \) and may return a good result of type \( T2 \) or fail.
Composition

Validate Request: T1 -> T2 eresult

Success T2

Read Request: T2 -> T3 eresult

Failure error

Goal: Create a bypass combinator to convert an ‘a -> ‘b eresult function into 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 validate1 input =  
  if input.name = "" then  
    Failure "no name"  
  else  
    Success input

let validate2 input =  
  if String.length (input.name) > 50 then  
    Failure "name too long"  
  else  
    Success input
Using the bypass combinator

let validate1' = bind validate1
let validate2' = bind validate2

let validate1 : request -> request eresult
let validate2 : request -> request eresult

let (>>) f g x = g (f x)

let validator = validate1'
                  >> validate2'
                  >> validate3'

validator : request eresult -> request eresult

(>>) : ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)
let (>=>) f1 f2 =
  fun x ->
    match f1 x with
    Success s -> f2 s
  | Failure f -> Failure f

validator =
  validate_name1
  >>= validate_name2
  >>= validate_email

validator : request -> request eresult

similar to ordinary function composition, but for eresults
An Error-Processing Library

(|>) : ‘a -> (‘a -> ‘b) -> ‘b

(* generic pipe *)

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

(* generic function composition *)

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

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

return : ‘a -> ‘a eresult

(* successful with ‘a *)

fail : string -> ‘a eresult

(* automatic failure *)

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

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

(* convert an error-free function *)

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

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

(|>) : ‘a -> (‘a -> ‘b) -> ‘b

(* generic pipe *)

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

(* generic function 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 triple of \((\text{set of values, bind, return})\) that satisfies certain equational laws:

\[
\text{return } a \gg=} f) \equiv f a \\
m \gg=} \text{return} \equiv m \\
m \gg=} (\text{fun } x \to k x \gg=} h) \equiv m \gg=} k \gg=} h
\]

- 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 \(a \to b\) has no effect. Only functions with type \(a \to M b\) for certain monads \(M\) have effects.
Functional programming is awesome.

It is no fun to build libraries like this in Java:

**SCORE:** OCAML 4, JAVA 0