Session 9: Type-driven design

COMP2221: Functional programming

Lawrence Mitchell*

*lawrence.mitchell@durham.ac.uk

Introduction

provide a nice "surface" to users of code.

- Haskell offers easy use of quite sophisticated types
- Will discuss some ways of thinking about API design
- Goal is to think about APIs that enforce compile-time correct use

$\Rightarrow$ influence the way you write code in all languages

$\Rightarrow$ aim of minimise bugs.

Correct merging?

Spot the bug

```
mergeBy :: (a -> a -> Ordering) -> [a] -> [a] -> [a]
mergeBy _ [] ys = ys
mergeBy _ xs [] = xs
mergeBy cmp (x:xs) (y:ys)
  | cmp x y == LT = x : mergeBy cmp xs (y:ys)
  | otherwise     = y : mergeBy cmp (x:xs) ys
```

Correct merging?

Spot the bug

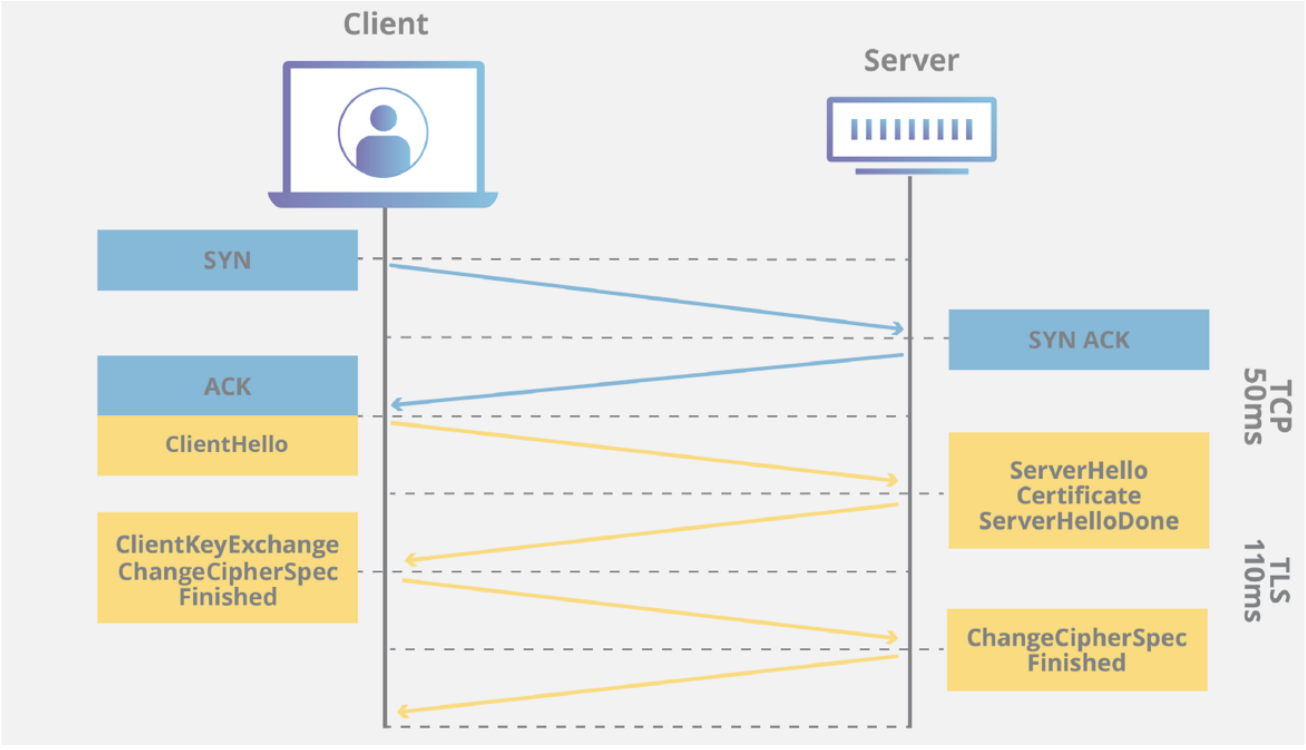
```
mergeBy :: (a -> a -> Ordering) -> [a] -> [a] -> [a]
mergeBy _ [] ys = ys
mergeBy _ xs [] = xs
mergeBy cmp (x:xs) (y:ys)
  | cmp x y == LT = x : mergeBy cmp xs (y:ys)
  | otherwise     = y : mergeBy cmp (x:xs) ys
```

- Only correct if `xs` and `ys` were both sorted using `cmp`!

⇒ would like incorrect calls to fail at compile time.
⇒ could check at runtime, destroys correct complexity. :C

Secure web connections? I

TLS handshake



https://www.cloudflare.com/en-gb/learning/ssl/what-happens-in-a-tls-handshake/

Secure web connections? II

TLS handshake for web security: RSA key exchange

1. The ‘client hello’ message: [...]. The message will include [...], and a **string of random bytes** known as the “client random.”
2. The ‘premaster secret’: The client sends **one more random string of bytes**, [...] **encrypted with the server’s public key** [...]

⇒ Conditions for correct use of protocol.

Secure web connections? II

TLS handshake for web security: RSA key exchange

1. The ‘client hello’ message: [...]. The message will include [...], and a **string of random bytes** known as the “client random.”
2. The ‘premaster secret’: The client sends **one more random string of bytes**, [...] **encrypted with the server’s public key** [...]

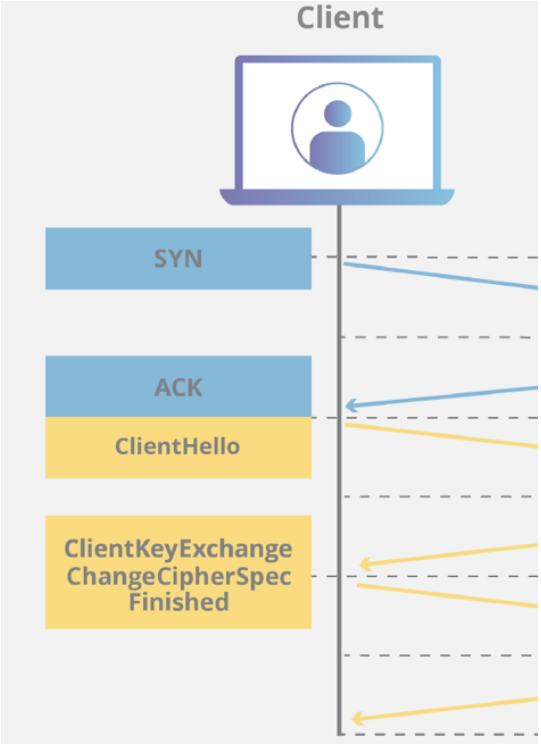
What if we forget these things?

What might an API look like?

Simple Python API

```
def open(address):
    return open_socket(address)
def receive(socket, n):
    return socket.read(n)
def send(socket, msg):
    return socket.write(len(msg), msg)
```

A first go




```
s = open(address)
s = send(s, "syn")     # syn
ack = receive(s, _)    # ack
# Send hello
s = send(s, "hello" + random())
# Get server cert
cert = receive(s, _)
s = send(s, "secret") # oops!
```

What went wrong?

- Our API has no way of enforcing valid state
- Typical approach to solve this: sprinkle some assertions/validation through the code

⇒ *antipattern* since can easily forget things

"code runs correctly without assertions"

```
assert :: Data -> ()
```

What went wrong?

- Our API has no way of enforcing valid state
- Typical approach to solve this: sprinkle some assertions/validation through the code

$\Rightarrow$ *antipattern* since can easily forget things

Better approach

Build the state into the type system, only implement methods on states that allow them.

The TLS handshake again

```
class Conn:
  def send_hello(self):
    return OpenConn(self.sock,
                    self.sock.send(...))
class OpenConn:
  def receive_cert(self):
    return ConnWithCert(self.sock,
                        self.sock.recv(...))
class ConnWithCert:
  def send_premaster(self):
    return ConnWithPremaster(self.sock,
                             self.sock.send(...))
```

```
conn = Conn(open(address))
         .send_hello()
         # API requires we
         # call this
         .receive_cert()
         # before calling this
         .send_premaster()
```

Moral

- In Python incorrect method chaining will only be caught at runtime
- ...still better than security holes!
- Idea is to encode *state* of program in the *types*
- In statically-typed languages this can be caught at compile time.

This method-chaining pattern is a very popular design pattern called a *fluent interface*.

You've doubtless seen it in any javascript library you've used.

Parse, don't validate

- Another place where type-driven design arises is consuming "unstructured" data from the outside world and turning it into something structured
- Prototype might be stream of bytes into JSON
- Two broad options for checking "invalid" data
  1. validation: assert data are well-formed (as side-effect)
  2. parse-and-continue: assert data are well-formed and return new type

What's the difference?

- Validation `validate :: SomeData -> ()` can be elided
- Parsing `parse :: Unstructured -> Structured` cannot

⇒ the conclusions of validation "these data are now valid" cannot be encoded in the type

- Can't guarantee downstream correctness

Prototype: a safe head

```haskell
safeHead :: [a] -> Maybe a
safeHead (x:_) = Just x
safeHead _ = Nothing
-- Or

data NonEmpty a = Cons a [a]

nonEmpty :: [a] -> Maybe (NonEmpty a)
nonEmpty [] = Nothing
nonEmpty (x:xs) = Just (NonEmpty x xs)

nonEmptyHead :: NonEmpty a -> a
nonEmptyHead (Cons x _) = x
```

What's the difference

- Suppose we are parsing a list which might be empty, and want to check that case and then pass it on.
- `nonEmpty` constructor does the checking, and then delivers a type that is provably non-empty

$\Rightarrow$ don't need to check again!

- `safeHead` approach forces us to always check (because we only have a `[a]`)

Moral

Encode *refinements* from validation in the types.

Any check that is required to pass for a program to proceed with valid data should not be a "side condition".

Back to merging

```
mergeBy :: (a -> a -> Ordering) -> [a] -> [a] -> [a]
mergeBy _ [] ys = ys
mergeBy _ xs [] = xs
mergeBy cmp (x:xs) (y:ys)
  | cmp x y == LT = x : mergeBy cmp xs (y:ys)
  | otherwise     = y : mergeBy cmp (x:xs) ys
```

The bug here is rather hard to handle. Want a type

```
mergeBy ::
   ({a -> a -> Ordering} cmp)  -- Name this parameter
   -> SortedBy cmp [a]
   -> SortedBy cmp [a]
   -> SortedBy cmp [a]
```

→ list + proof that sorted by cmp.

This is *just about possible* in Haskell 2010, need more sophisticated types than what we've seen (see `https://kataskeue.com/gdp.pdf` if you're keen)

→ builds just enough dependent types

Concluding remarks

- This is a somewhat philosophical set of slides
- I think that thinking about types and the invariants they capture is a good way to design APIs.
- If you do this, you will be better than 99% of web framework developers. ✓
- Many places to go for further reading, ideas here, these are some nice ones
  - Parse, don't validate `https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/`
  - Type state patterns `http://cliffle.com/blog/rust-typestate/`
  - Ghosts of departed proofs `https://kataskeue.com/gdp.pdf`
  - An introduction to formal methods and proof automation `https://dependenttyp.es/classes/598sp2022.html`