WEB3, Session 8

Functional Programming Patterns

Ole I Hougaard, olehougaard@gmail.com

Problem 1: Control Flow

```
const dragons = (pets: Pet[]) =>
  pets.filter(p => p.type === 'dragon')
const ages = (pets: Pet[]) =>
  pets.map(p => p.age)
const sum = (ns: number[]) =>
  ns.reduce((sum, n) => sum + n, 0)

let sumOfAgeOfDragons = sum(ages(dragons(pets)))
```

control flow

Problem 1: Control Flow - early return

```
export function player_in_turn(games: Game[], id: number) {
    const game = games.find(g => g.id === id)
    if (game === undefined) return undefined
    if (game.pending) return undefined
    if (game.player_in_turn === undefined) return undefined
    return game.players[game.player_in_turn]
}
```

Problem 1: Control Flow - callback hell (I)

```
function games_waiting_for_player(player, callback) {
  const xhr = new XMLHttpRequest()
  xhr.open('GET', 'http:/localhost:8080/active')
  xhr.onload = _ => {
    const games = JSON.parse(xhr.responseText)
    const active_games = games
      .filter(g => g.player_in_turn !== undefined)
    const waiting_games = active_games
      .filter(g => g.players[g.player_in_turn] === player)
```

Problem 1: Control Flow - callback hell (II)

```
        const loaded_games = []
        for(let game of waiting_games) {
            const gameXhr = new XMLHttpRequest()
            gameXhr.open('GET', `http://localhost:8000/active/${game.id}`)
            gameXhr.onload = _ => {
                loaded_games.push(JSON.parse(gameXhr.responseText))
                if (loaded_games.length === waiting_games.length)
                    callback(loaded_games)
            }
            gameXhr.send()
        }
    }
    xhr.send()
```

Problem 2: Proliferation of anonymous functions

```
const sumOfAgeOfDragons = pets
  .filter(p => p.type === 'dragon')
  .map(p => p.age)
  .reduce((acc, age) => acc + age, 0)
```


Ole I Hougaard, olehougaard@gmail.com

Problem 3: Updating data

```
const hireEmployee =
  (e: Person, c: Company) =>
    createCompany(
      c.name,
      c.address,
      [...c.employees, e])
```

{
name: c.name,
address: c.adress,
employees: [...,e]
}

{...c, employees: [...c.employees, e]}

Problem 4: Excessive array creation

```
const sumOfAgeOfDragons = pets                  – an array
   .filter(p => p.type === 'dragon')            – returns new array
   .map(p => p.age)                             – returns new array
   .reduce((acc, age) => acc + age, 0)
```

Solutions

- (1) Monads and/or functional composition
- (2) Currying
- (3) Persistent data structures
  - Allow changes with minimal copying
- (4) Lazy sequences/Flyweight
  - Intermediate object delays construction of array until necessary

Functors and Monads

- Functors and monads are data structures/objects that are designed for pipelines
- They are types that wrap around another "element" type
  - like Array<string>
- The point of the functors and monads is that you can apply functions on the "elements"
- The result with be the same type of functor or monad, but with the value updated

Functors

- Has a function/method that allows application of simple functions to the data inside
  - The function/method is usually called *map*
- Assume we have a functor, F<T>
  - The functions we can apply would have type (t: T) => U
  - The result of mapping the functor would have type F<U>
- Arrays are ~~monads~~ functors
  - const a: Array<string> = ['here', 'we', 'go']
  - s => s.length has type (s: string) T => number U
  - a.map(s => s.length) is an Array<number>

Monads (usu. also functors, usu. has filter)

- Monads: Has a function/method that allows for deconstruction, application of a function, then reconstruction of the data structure
  - The function/method is usually call *flatMap* (formally "bind")
- Assume we have a monad, `M<T>`
  - The functions we can apply have type `(x: T) => M<U>` (Sometimes also (x: T) => U)
  - The result of mapping the monad will have type `M<U>`
- Arrays are monads because of the flatMap method
  - `const a: Array<string> = ['modern', 'condition', 'fix']`
  - `s => ['pre'+s, 'post'+s]` has type `(s: string) => Array<string>`
  - `a.flatMap(s => ['pre'+s, 'post'+s])` has type `Array<string>`

The Promise Monad

```
fetch('http:/localhost:8080/active')
   .then(res => res.json())
   .then(games =>
        games.filter(g => g.player_in_turn !== undefined))
   .then(active_games =>
        active_games.filter(g =>
            g.players[g.player_in_turn] === player))
   .then(waiting_games =>
       waiting_games.map(g =>
           fetch(`http://localhost:8000/active/${g.id}`)))
   .then(Promise.all)
   .then(callback)
```

returns Promise

doesn't return Promise

returns Promise

.then(ress => ress.map(res => res.json()))

.then (Promise.all)

The Optional Monad

- Also known as Option or Maybe
- Alternative to T | undefined (or T | null)
- Optional<T>: The element is there, or it isn't
- Similar to an array with at most 1 element

Optional interface

```
export interface Optional<T> {
  isPresent(): boolean
  ifPresent(consumer: (element: T) => void): void
  map<U>(f: (element: T) => U): Optional<U>
  flatMap<U>(f: (element: T) => U | Optional<U>): Optional<U>
  filter(predicate: (element: T) => boolean): Optional<T>
  get(): T
  getOrElse(fallback: T): T
  or(other: Optional<T>): Optional<T>
}
```

Optional implementation

- There are no "standard" optional libraries
- Might be because TypeScript offers enough protection against nullish
- Implemented it myself
- See
  - `"08 Functional Patterns/patterns/optional.ts"`
  - `"08 Functional Patterns/patterns/with_optional.ts"`

Either Monad

- When a function can return ~~to~~ 2 very different things
- Alternative to T | U
- Often used to return errors from a function: Either<Error, ~~|~~ Response>
- In that case it is often called a Try or a Result monad.
- See "yahtzee functional/server/src/response.ts"

lodash

legal JS variable name

- import * as _ from 'lodash'
- Many extra utility functions
- Mostly for functional programming, but some functions mutate their input
- lodash/fp is purely functional
  - import * as _ from 'lodash/fp'
- We'll mostly look at lodash/fp

"Modifying" (lodash/fp)

- _.set
  - `_.set(['scores', key], score(…), section)`
  - Corresponds to `section.scores[key] = score(…)`

Path

Value

Object

returns new object

- Instead of _.set we can apply a function: _.update
  - `const addEmployee = (e: Person, c: Company) =>`
    `_.update('employees', _.extend(e), c)`

function that adds e at the end.

Currying

- A curried function is a function that takes several parameters "one at a time"
- It gets one parameter and returns a new function
- That function gets another parameter and so on until all parameters are there
- The last function returns a value
- Used for *partial evaluation*
- In some languages, all functions are curried: F#, Haskell, Elm,…

Curried add

```
const numbers = [2, 3, 7, 11, 13]

const addCurry = (a: number) => (b: number) => a + b
```

Returns a function

```
// Good for
console.log(numbers.map(addCurry(5)))
```

//[7, 8, 12, 16, 18]

instead of numbers.map(x => x + 5)

```
// Bad for
console.log(addCurry(7)(11))
```

Curried add (2)

```
function add(a: number): (b: number) => number
function add(a: number, b: number): number
function add(a: number, b?: number) {
  if (b !== undefined)
    return a + b
  return (b: number) => a + b
}

console.log(numbers.map(add(5)))
console.log(add(7, 11))
```

overloaded types

implementation

Control flow - variables vs flow

~~x => f(x)~~

f

```
const sumOfAgeOfDragons = (pets: Pet[]) => {
  const dragonPets = dragons(pets)
  const dragonAges = ages(dragonPets)
  return _.sum(dragonAges)
}
```

flow of control

lodash

```
const sumOfAgeOfDragons2 = (pets: Pet[]) => _.flow([
    dragons,
    ages,
    _.sum
  ])(pets)
```

flow of control

_.flow([...])

is a function

Point-free style

– Optional, probably too hard to read

```
const sumOfAgeOfDragons3 = _.flow([
    dragons,
    ages,
    _.sum
  ])
```

lodash/fp functions are curried

(pets =>

pets

```
const sumOfAgeOfDragons2 = _.flow([
    _.filter((pet: Pet) => pet.type === 'dragon'),
    _.map((pet: Pet) => pet.age),
    _.sum
  ])
```

(pets)

Using curried utility functions

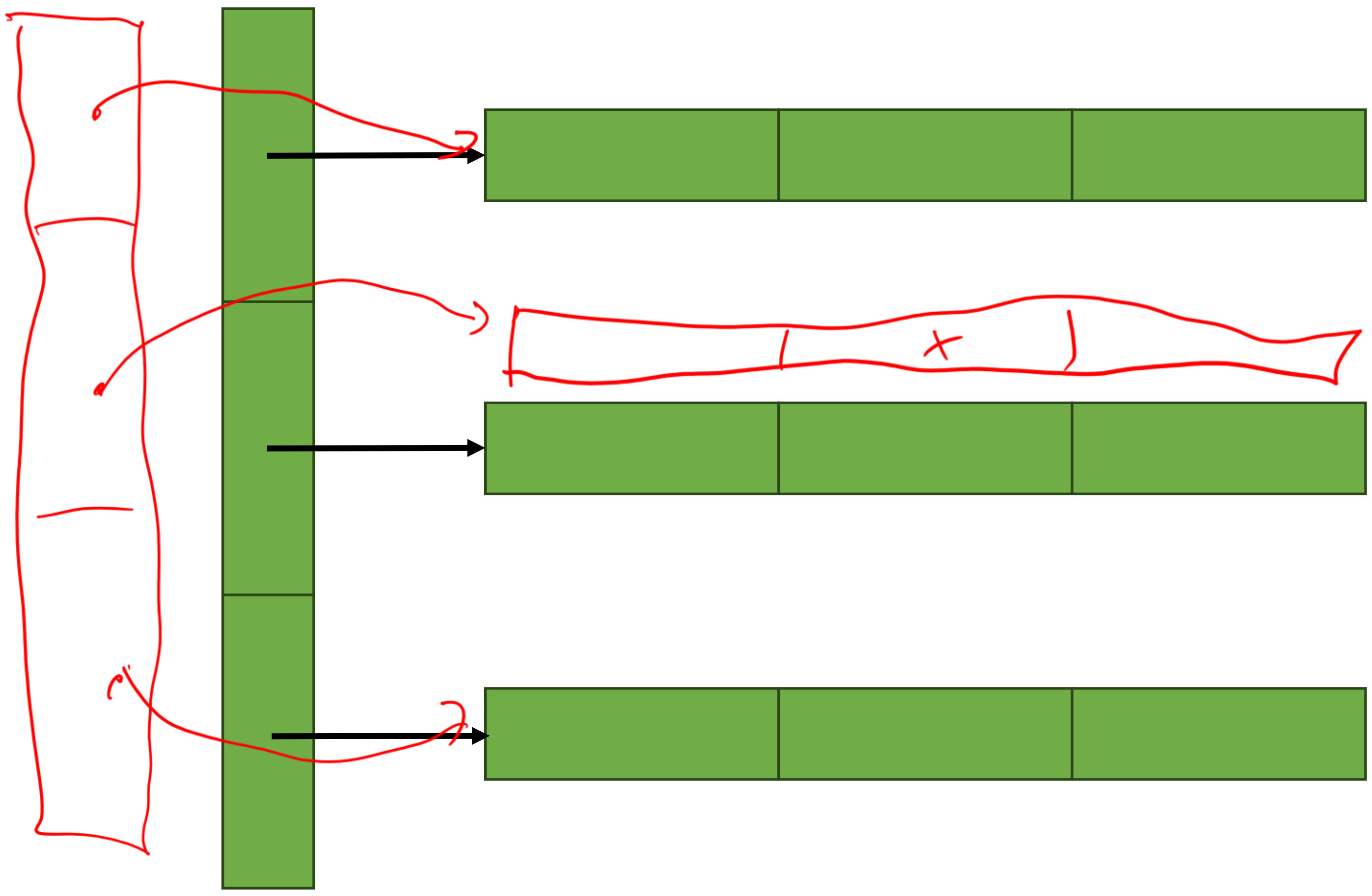
```
const sumOfAgeOfDragons3 = (pets: Pet[]) => _.flow([
    _.filter(_.matches({type: 'dragon'})),
    _.map(_.prop('age')),
    _.sum
])(pets)
```

(where)

(select)

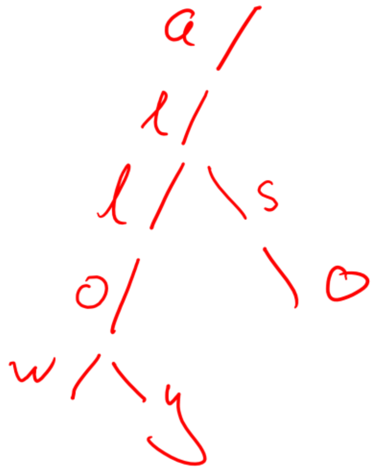
flow of control

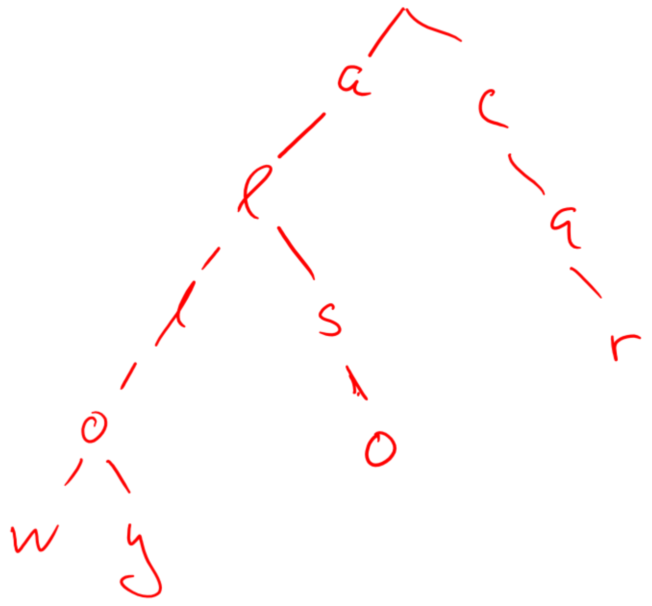
Reducing copying: tic-tac-toe



The Trie

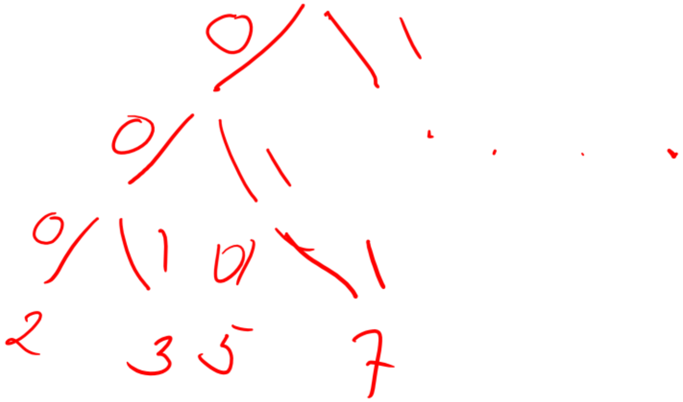
Store 'allow', 'alloy', 'also'





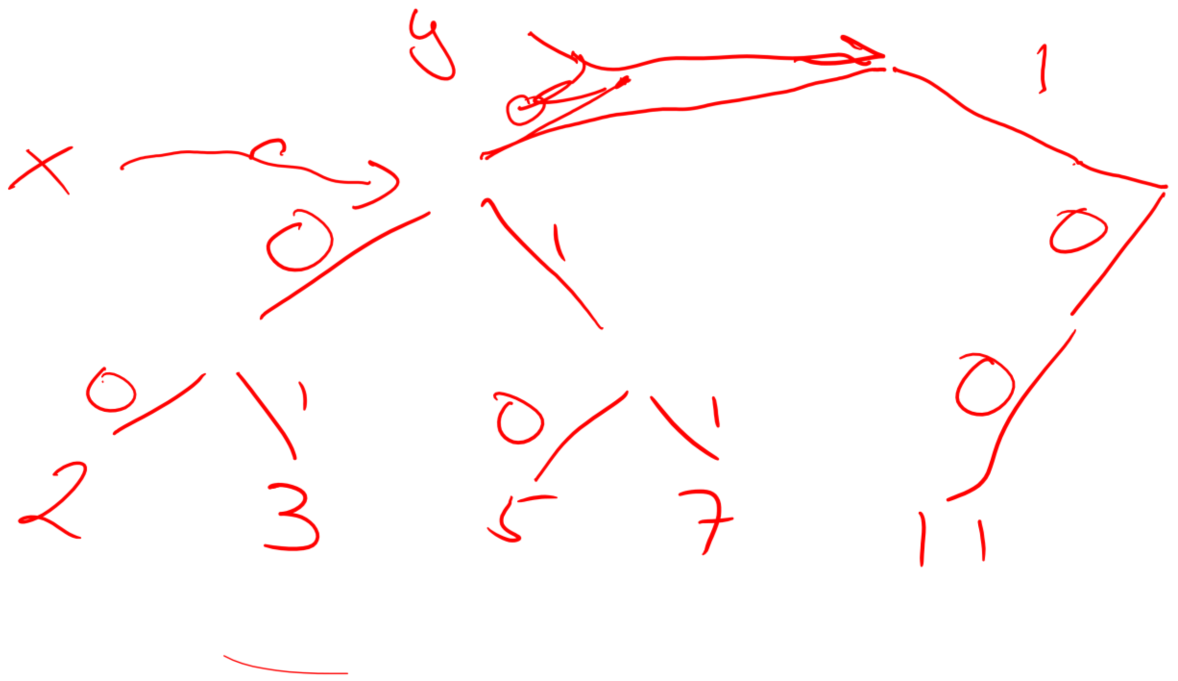
The Binary Trie

[2, 3, 5, 7, 11, 13, 17, 19]



"Adding" an element

$y = x.push(11)$

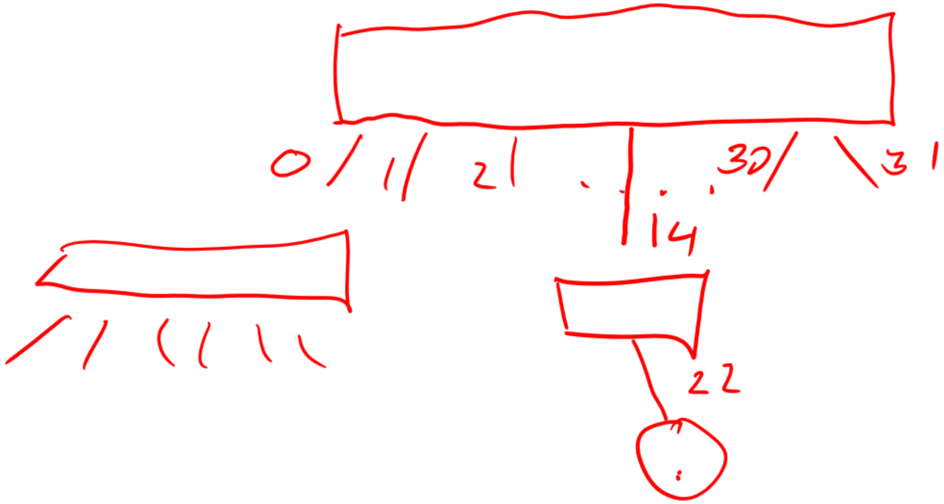


Every operation is $O(\log n)$

Increasing the fanout

$$\text{Index} = \overbrace{01110}^{5\text{ bits}}\ \overbrace{10110}^{5\text{ bits}}$$

14 22



immutable.js

- List
  - Replaces arrays
- Map
  - An associative array ~ Same value type
- Set
  - Distinct data
- Seq
  - Lazy sequence
- fromJS
  - A function to turn your data structure immutable
  - Warning: your objects become Maps

List

Standard TypeScript

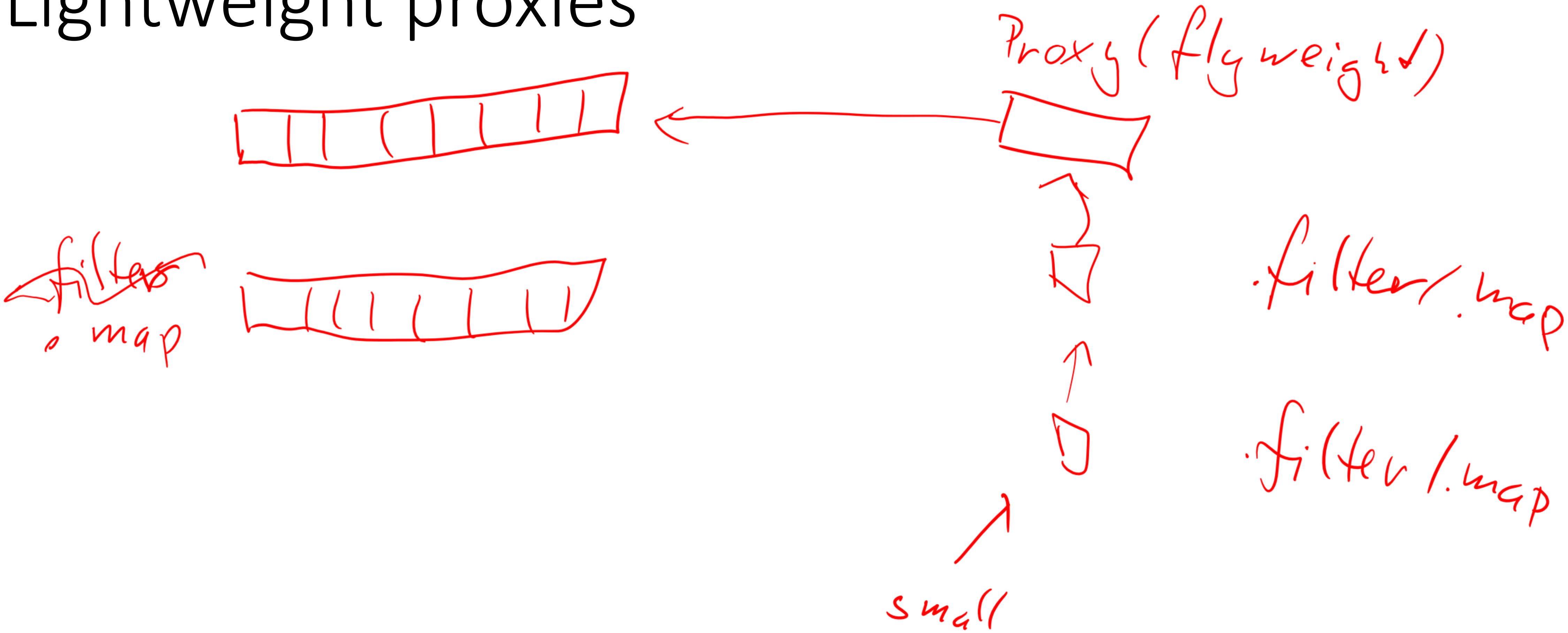
```
const hireEmployee =
  (e: Person, c: Company) =>
    createCompany(
      c.name,
      c.address,
      [...c.employees, e])
```

With immutable

```
addEmployee(employee: Person):
Company {
  return new Company(
    this.name,
    this.address,
    this.employees.push(employee))
}
```

non-destructive
returns new list

Lightweight proxies



Array creation

Standard TypeScript

```
const sumOfAgeOfDragons = pets
  .filter(p => p.type === 'dragon')
  .map(p => p.age)
  .reduce((acc, age) => acc + age, 0)
```

With immutable

Proxs

```
const sumOfAgeOfDragons3 = Seq(pets)
  .filter(p => p.type === 'dragon')
  .map(p => p.age)
  .reduce((sum, age) => sum + age, 0)
```

No new arrays

lodash (not lodash/fp)

lodash

```
_.chain(pets)
  .filter(p => p.type === 'dragon')
  .map(p => p.age)
  .reduce((sum, a) => sum + a, 0)
  .value()
```

immutable.js

```
const sumOfAgeOfDragons3 = Seq(pets)
  .filter(p => p.type === 'dragon')
  .map(p => p.age)
  .reduce((sum, age) => sum + age, 0)
```

Other languages

Java

```
pets.stream()
  .filter(p->p.type().equals("dragon"))
  .mapToInt(p -> p.age())
  .sum();
```

immutable.js

```
const sumOfAgeOfDragons3 = Seq(pets)
  .filter(p => p.type === 'dragon')
  .map(p => p.age)
  .reduce((sum, age) => sum + age, 0)
```

Alternative in Java

Lambda expression

```
pets.stream()
    .filter(p ->
        p.type().equals("dragon"))
    .mapToInt(p -> p.age())
    .sum();
```

Method reference

```
pets.stream()
    .filter(p ->
        p.type().equals("dragon"))
    .mapToInt(Pet::age)
    .sum();
```

Lazy sequences instead of recursion

```
function factorial(n: number): number {
  return Range(1, n + 1).reduce((a, b) => a * b, 1)
}
```

Infinite sequences

```
function isPrime(n: number): boolean {
  return Range(2, Infinity)
    .takeUntil(i => i * i > n)
    .find(i => n % i === 0) === undefined
}

const first100Primes = Range(2, Infinity)
  .filter(isPrime)
  .take(100)
  .toJS()
```

Conclusion

- Use immutable.js to speed up updates to data structures
- Also to get more method to use
- Use Seq to speed up multiple transformations of lists

- Use lodash to think in functions
- And to get better methods for functional
- Beware the bus number