How to Lose FP at Work

If you're looking to lose functional programming at work, here are a bunch of mistakes I've made on JS-heavy web teams over the years that can help you do the same! /s

```
web     :: rwp.im
github  :: rpearce
email   :: me@robertwpearce.com
tweeter :: @RobertWPearce
```

About me

- λ Work at **Articulate** (`articulate.com` / `rise.com`) doing web dev work (heaps of accessibility work, too)
- λ 12 years experience in the web world
- λ Enjoy Nix(⚙️), Haskell(λ), Rust(🦀), Elixir(🧪), and even Go(🐿️) on the side
- λ Was writing the Ramda Guide (`ramda.guide`), but life intervened, and I may strip it for parts
- λ Perpetual beginner-/intermediate-level FP practitioner
- λ Am a dad with another on the way!

Disclaimers/Heads up

λ This backwards-style talk will be sarcastic, snarky, and cringey

λ The examples are more JS-oriented, but the commentary is mostly universal

λ The examples are all *my* examples and personal work-related head-canon; this should not reflect poorly on my colleagues nor my employer

λ The slides aren't shouting at you — they're shouting at me

"Do"s and "Don't"s

Don't

have static type checking

λ No `TypeScript`

λ No `Flow`

λ No `ReasonML`

λ No `Elm`

λ No (insert language with static type checking that compiles to JS)

```
const processData = composeP(syncWithBackend, cleansePII, validateData)

// * What arguments and their types are expected here?
//
// * If each function is written like this, how can
//   one suss out what data are flowing where?
//
// * How hard is this going to be to debug?
//   Use this everywhere: `(x) ⟹ (console.log(x), x)`
```

So the point-free style is the problem? Not so fast…

```
async function processData(data) {
  await validateData(data)
  const cleansedData = cleansePII(data)
  await syncWithBackend(cleansedData)
  return data
}

// or for the Promise-chainers…

const processData = data ⟹
  validateData(data)
    .then(cleansePII)
    .then(syncWithBackend)
    .then(() ⟹ data)

// 👆 Are these any clearer? Y/n? ¯\_(ツ)_/¯
```

Don't

use well-known documentation tools

λ No `jsdoc`

λ …are there any other JS contenders?

Deprive your team of this clarity and helpful auto-completion:

```
/**
 * @typedef {Object} ReportingInfo
 * @property {("light"|"dark")} userTheme - Current user's preferred theme
 * @property {string} userName - Current user's name
 * @property {UUID} postId - The current post's ID
 */

/**
 * Validates that the reporting data (current user site prefences and post info)
 * is OK, removes personally identifiable information, syncs this info with the
 * backend, and gives us back the original data.
 *
 * @param {ReportingInfo} data - The current user's site preferences and post info
 * @returns {Promise<ReportingInfo>} - The original reporting data
 */
const processData = data ⟹ { /* … */ }
```

Don't

properly train new and existing colleagues

"Here, go read all these posts and books, watch these videos, and let me know if you have any questions!"

— *Me*

Don't

bother getting the other engineering teams on board and rowing  in the same direction

λ "If I build it, they will notice… right?" ❌

λ *Idea: Lunch 'n learn about FP?* ❌

λ *Idea: Meet with other team leaders?* ❌

Do

live by "Point-free or die"

@RobertWPearce | rwp.im | 2023-01-24 Auckland Functional Programming Meetup

"Think it's point-*less*?
Go watch `Point-Free or Die: Tacit Programming in Haskell and Beyond` by Amar Shah"

— *Me*

```
import { __, any, lt } from 'ramda'
const anyLt0 = any(lt(0, __)) // hint: this has a bug in it
anyLt0([1, 2, 3]) // true — ugh…

// vs. the probably pretty simple…

const anyLt0 = numbers ⟹ numbers.some(n ⟹ n < 0)
anyLt0([0, 1, 2, 3]) // false
anyLt0([0, 1, 2, -1, 3]) // true — looks good

//
//
//
```

```
import { __, any, lt } from 'ramda'
const anyLt0 = any(lt(0, __)) // hint: this has a bug in it
anyLt0([1, 2, 3]) // true — ugh…

// vs. the probably pretty simple…

const anyLt0 = numbers ⟹ numbers.some(n ⟹ n < 0)
anyLt0([0, 1, 2, 3]) // false
anyLt0([0, 1, 2, -1, 3]) // true — looks good

// 👆 should we resist eta-converting this?!
//
//
```

```
import { __, any, lt } from 'ramda'
const anyLt0 = any(lt(0, __)) // hint: this has a bug in it
anyLt0([1, 2, 3]) // true — ugh…

// vs. the probably pretty simple…

const anyLt0 = numbers ⟹ numbers.some(n ⟹ n < 0)
anyLt0([0, 1, 2, 3]) // false
anyLt0([0, 1, 2, -1, 3]) // true — looks good

// 👆 should we resist eta-converting this?!
// …
//
```

```
import { __, any, lt } from 'ramda'
const anyLt0 = any(lt(0, __)) // hint: this has a bug in it
anyLt0([1, 2, 3]) // true — ugh…

// vs. the probably pretty simple…

const anyLt0 = numbers ⟹ numbers.some(n ⟹ n < 0)
anyLt0([0, 1, 2, 3]) // false
anyLt0([0, 1, 2, -1, 3]) // true — looks good

// 👆 should we resist eta-converting this?!
// …
// NOT ON MY WATCH
```

```
const any = fn ⟹ array ⟹ array.some(fn)
const isLtN = x ⟹ n ⟹ x < n
const isLt0 = isLtN(0)
const anyLt0 = any(isLt0)

anyLt0([1, 2, 3]) // true — ugh; the bug is back
```

Real, but altered, example:

```
const finishItems = compose(
  flip(merge)({ isDone: true, amtComplete: 100 }),
  over(
    lensProp('indexedObjects'),
    mapVals(
      compose(
        over(lensProp('indexedObjects'), mapVals(assoc('isDone', true))),
        assoc('isDone', true)
      )
    )
  )
)
```

Do

prefer the wrong abstraction over the right duplication

"Prefer duplication over the wrong abstraction"

— *Sandi Metz' RailsConf 2014 talk,* `All the Little Things`

Instead…

1. Dilute core business logic to broad generalizations
2. Fail to understand category theory enough for this to be useful
3. Be the only one who knows how these abstractions work
4. Previously thorough PR-reviews now look like "👍"

Don't

refactor old patterns that clearly don't work for the team

Do

force functional patterns into a language that wasn't built for them

- λ Recursive functions
  - λ Handle with `trampolines` if you really want them
- λ Cryptic stack traces thanks to currying and composing functions
  - λ `Debugging functional` by Brian Lonsdorf
  - λ `Partially-applied (or curried) functions could obfuscate the JavaScript stack trace` by Thai Pangsakulyanont
- λ No GHC-style fusing of `.map(…).map(…).map(…)`
- λ BYO algebraic data type libraries (they're well done, though)

Do

opaquely compose and sequence the entirety of your API endpoints and make them hard to debug

On the surface, this isn't so difficult to read…

```
// handler for POST /posts
import { createPost } from 'app/db/posts'
import { authenticateUser, authorizeUser } from 'app/lib/auth'
import { trackEvent } from 'app/lib/tracking'

const validateRequestSchema = payload => { /* … */ }

export const handleCreatePost = curry(metadata =>
  pipeP(
    authenticateUser(metadata),
    authorizeUser(metadata),
    validateRequestSchema,
    createPost(metadata),
    tapP(trackEvent('post:create', metadata)),
    pick([ 'id', 'authorId', 'title' ])
  )
)
```

Did you catch or wonder about these?

- λ `handleCreatePost` expects 2 arguments?
- λ `authenticateUser` ignores the 2nd curried parameter sent to it? How would you?
- λ Does `trackEvent` receive the `payload` passed through or the result of the `createPost()` fn?

Let's try something else…

```
export async function handleCreatePost(metadata, payload) {
  await authenticateUser(metadata)
  await authorizeUser(metadata, payload)
  await validateRequestSchema(payload)

  const post = await createPost(metadata, payload)

  await trackEvent('post:create', metadata, post)

  return {
    id: post.id,
    authorId: post.authorId,
    title: post.title,
  }
}
```

✅ Not forcing different arity functions into a pipeline pattern

💡 But if you want to make things trickier for people, go with the first approach

Do

recreate imperative, procedural programming while calling it "declarative"

```
const setBookReadPercentByType = (contentType, statusObject) ⟹
  assoc(
    'readPercent',
    pipe(
      prop('subItems'),
      values,
      filter(propEq(contentType, 'chapter')),
      length,
      flip(divide)(compose(length, keys, prop('subItems'))(statusObject)),
      multiply(100),
      Math.round
    )(statusObject),
    statusObject
  )
```

Do

have 8+-ish different patterns for function composition

👇 These 4, plus Promisified versions of each, plus combinations of them all used at once; doesn't include ramda's even more abstract `composeWith` and `pipeWith`

```
// compose (plus composeP for Promises)
const getHighScorers =
  compose(
    mapProp('name'),
    takeN(3),
    descBy('score')
  )

// pipe (plus pipeP for Promises)
const getHighScorers =
  pipe(
    descBy('score'),
    takeN(3),
    mapProp('name')
  )
```

```
// composeWithValue
const getHighScorers = players =>
  composeWithValue(
    mapProp('name'),
    takeN(3),
    descBy('score'),
    players
  )

// pipeWithValue (plus pipePWithValue for Promises)
const getHighScorers = players =>
  pipeWithValue(
    players,
    descBy('score'),
    takeN(3),
    mapProp('name')
  )
```

Do

make yourself one of the few who can debug algebraic data types during midnight incidents

Ensure your team is surprised by all of the following words when debugging or altering your code in the pursuit of their own tasks:

- λ `Task`, `Maybe`, `Either`, `Result`, `Pair`, `State`
- λ `bimap`
- λ `chain`
- λ `bichain`
- λ `option`
- λ `coalesce`
- λ `fork`
- λ `sequence`
- λ `ap`
- λ `map` — and I don't mean `Array.prototype.map`, nor a `new Map()`, nor a key/value object

Do

suggest, on PRs, that colleagues completely refactor what they've done to fit your functional style

"**What you have here works great**, but what could this look like if we flipped all the function arguments around, removed all these intermediate variables and `if/else if/elses`, and mapped these operations over an `Either`?"

— Me

"I noticed you're explicitly constructing these objects in their functions. **If you were to use <UTILITY-FUNCTION>**, you could declare the shape of your outputted object and use functions as the values to look up or compute each value given some data."

— Me

Ok, last ones!

Do

sow imposter syndrome in others and exclude them by sharing non-beginner FP articles

Do

Do keep writing code using FP tools even when nobody else on the team is

Do

achieve peak perceived passive-aggression by getting tired and commenting PRs with emojis

Do

have "the FP talk" at work, and then publicly own your mistakes

Real-talk takeaways

We could write all this off as symptoms of:

- λ Inexperience
- λ Lack of technical leadership from me
- λ *Obviously* not the right paths — so incompetence?
- λ I hope not; I think it's deeper

Most things in life need to be tended to

λ our relationships 💜

λ our mental and physical health 🌈 🏃

λ our gardens 🪴

Paths can be accidentally created, too

Some issues and failures that got me here:

1. Persistent imposter syndrome
2. Feeling I just need to ship features and look out for myself
3. Not taking responsibility for a path I helped create
4. Not tending to things that needed tending to

But all is not lost!

The core tenets of FP seem to remain:

- λ Immutability
- λ Purity
- λ Moving effects to the conceptual edge of an application
- λ Very few classes and inheritance (React & web components don't count), `map`/`filter`/`reduce`, etc.

The main point

Remember that the choices we do and don't make significantly shape our futures, so if you end up somewhere, make sure you got there on purpose.

That's it!

:q!

How to Lose FP at Work

```
web      :: rwp.im
github   :: rpearce
email    :: me@robertwpearce.com
tweeter  :: @RobertWPearce
```