Functional Programming

Chris Constable

"Why write a function to solve a problem, when you can write a function which returns a function to solve that problem?"

"The functional programmer sounds rather like a medieval monk, denying themselves the pleasures of life in the hope that it will make them virtuous." - John Hughes

The problem of complexity

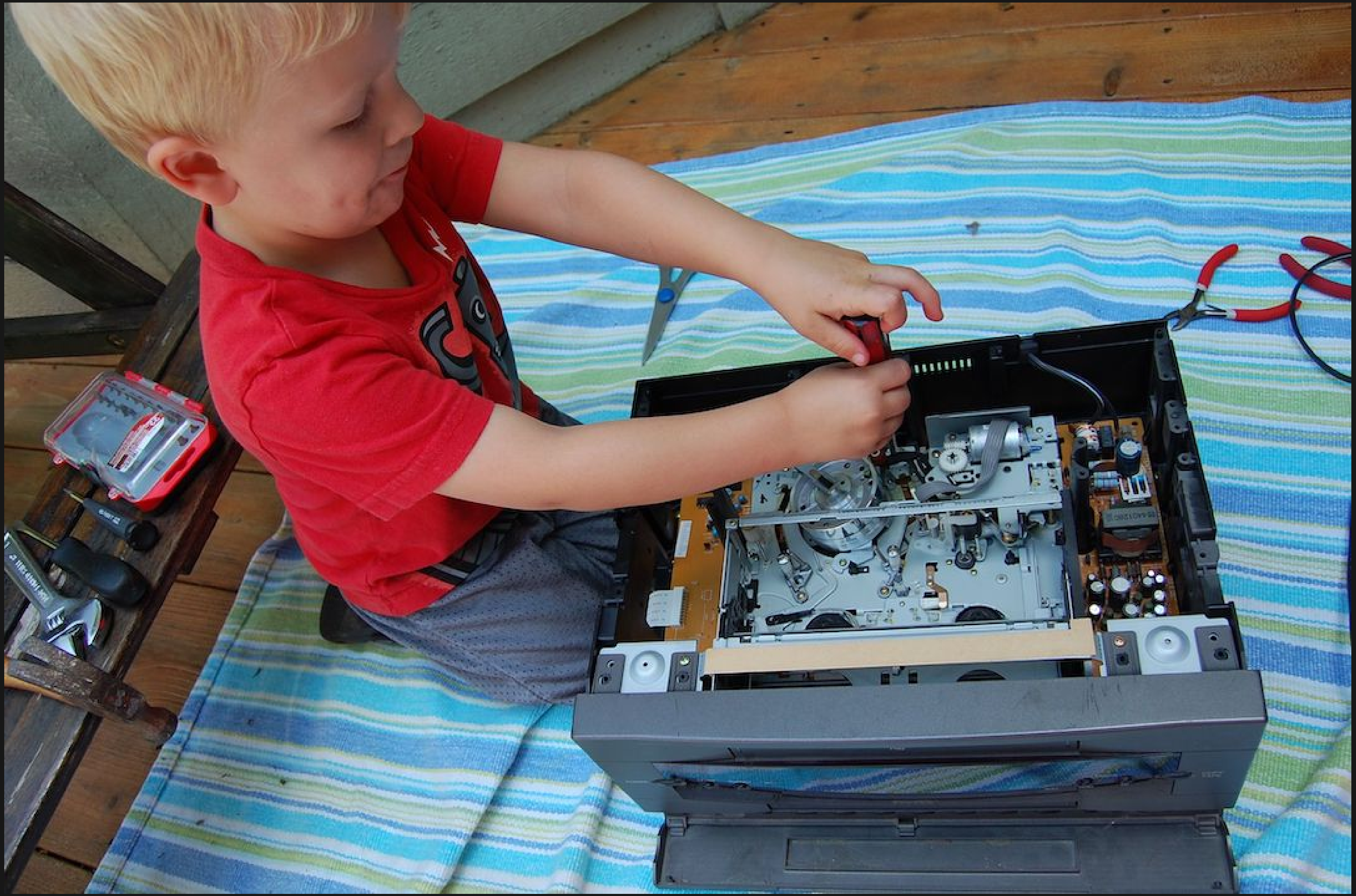
The problem of complexity

- Software is complex because life is complex

The problem of complexity

- Software is complex because life is complex
- How do we deal with complexity?

We break big problems into smaller ones



The problem of complexity

- We are taught to decompose problems into smaller, easier ones

The problem of complexity

- We are taught to decompose problems into smaller, easier ones
- This is great, *but it's only half the story*.

How we put solutions together is just as important as how we break problems down

Functional Programming

- Functional programming (FP) is a programming style that helps us *break problems down* and *compose solutions together*.

Functional Programming

- Functional programming (FP) is a programming style that helps us *break problems down* and *compose solutions together*.
- FP provides powerful mechanisms for *controlling complexity*.

Functional Programming

- Functional programming (FP) is a programming style that helps us *break problems down* and *compose solutions together*.
- FP provides powerful mechanisms for *controlling complexity*.
- FP *gives us tools* to create *modular* and *composable* code.

Modules

Modules

- "something that can be reused"
- "something self-contained"
- "isolated"
- "does one general thing"
- "consistently works"
- "black box"

module == function

Modularity

Modularity

- Modules are just functions

Modularity

- Modules are just functions
- These functions have properties:
  - *Deterministic*: Given the same input we get the same output ("mathematical functions")
  - *No free variables*: Functions don't depend on external constants, system calls, or their environment
  - *No "side-effects"*: Executing one function should not change how another function executes

Modularity

- Modules are just functions
- These functions have properties:
  - *Deterministic*: Given the same input we get the same output ("mathematical functions")
  - *No free variables*: Functions don't depend on external constants, system calls, or their environment
  - *No "side-effects"*: Executing one function should not change how another function executes
- Functions that have all these properties are called *pure functions*.

> Exercise 1

Write a *pure function* that takes a list of numbers, adds one to each number, and then returns the sum of the incremented numbers.

```
f([1])        == sum of [2]            == 2
f([1, 2])     == sum of [2, 3]         == 5
f([9, 9, 9])  == sum of [10, 10, 10]   == 30
```

> Exercise 1 Solution

```
function exercise1(list) {
  var sum = 0
  for (e of list) {
    sum += (e + 1)
  }
  return sum
}
```

> Exercise 2

Write a *pure function* that takes a list of numbers, adds one to each number, and then returns the sum of the incremented numbers.

- Do not use using any loop constructs e.g `for`, `while`, etc

🤔

Composition

To solve the previous exercise we need to introduce some "glue" we can use to compose modules.

Composition

`map`

Composition

`map`

Think of `map` as something that *transforms* all the elements of some *container-y* structure by using a given function and returns a new structure.

Composition

`map`

Think of `map` as something that *transforms* all the elements of some *container-y* structure by using a given function and returns a new structure.

```
[1, 2, 3].map(x => x + 1) == [2, 3, 4]
[1, 2, 3].map(x => x * x) == [1, 4, 9]
[1, 2, 3].map(x => x)     == [1, 2, 3]
[1, 2, 3].map(x => 5)     == [5, 5, 5]
```

Composition

Pitfall: Don't mistake `map` as something unique to arrays or hashmaps or "lists". It is a generic idea!

Composition

`fold`

Composition

`fold`

`fold` is a function that takes a *traversable data structure* (like an array), *a function for combining* two elements of that data structure, and *an initial value*, and returns the result of "folding" all the elements into each other until only one element is left.

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

```
0
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

```
0
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

```
add(0, 1)
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

```
1
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

```
1
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

```
add(1, 2)
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

```
3
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

```
3
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

```
add(3, 3)
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 0, [1, 2, 3])  // == 6
```

```
6
```

fold

```
const add = (x, y) => { return x + y }
fold(add, 100, [1, 4, 10])  // == 115
```

Composition

`fold` is sometimes called `reduce`, `accumulate`, `aggregate`, `compress`, or `inject`

Composition

Fold in Javascript

```
const add = (x, y) => { return x + y }
[1, 2, 3].reduce(add, 0)  // == 6
[1, 4, 10].reduce(add, 0) // == 15
```

Fold can do many things

```
const sum          = fold(+, 0)
const product      = fold(*, 1)
const concatList   = fold(+, [])
const concatString = fold(+, "")
const allAreTrue   = fold(&&, true)
const anyIsTrue    = fold(||, false)
```

> Exercise 2

Write a program that takes a list of numbers, adds one to each number, and then returns the sum of the incremented numbers.

- Do not use using any loop constructs e.g `for`, `while`, etc

> Exercise 2 Solution

```
function exercise2(list) {
  const increment = x => x + 1
  const add = (x, y) => x + y
  return list.map(increment).reduce(add, 0)
}
```

> Exercise 2 Solution

```
function exercise2(list) {
  const increment = x => x + 1
  const add = (x, y) => x + y
  return list.map(increment).reduce(add, 0)
}
```

Notice the last line is just function calls: `map`, `increment`, `reduce`, and `add`.

> Exercise 2 Solution

```
function exercise2(list) {
  const increment = x => x + 1
  const add = (x, y) => x + y
  return list.map(increment).reduce(add, 0)
}
```

We were *guided* to create functions for handling the individual, smaller problems: incrementing and adding.

> Exercise 2 Solution

```
function exercise2(list) {
  const increment = x => x + 1
  const add = (x, y) => x + y
  return list.map(increment).reduce(add, 0)
}
```

Finally, we use some FP *"glue"* to *compose* the solutions together: `map` and `reduce`.

> Exercise 2 Solution

```
function exercise2(list) {
  const increment = x => x + 1
  const add = (x, y) => x + y
  return list.map(increment).reduce(add, 0)
}
```

Reminder: `map` and `reduce` do not mutate the original array! They produce new arrays.

We defined solutions with small pure functions and glued (composed) them together

Recap: Modularity

Recap: Modularity

- Functions are building blocks and are treated like variables

Recap: Modularity

- Functions are building blocks and are treated like variables
- Functions can take other functions as input and return other functions as output
    - These are called *higher-order functions*

Recap: Modularity

- Functions are building blocks and are treated like variables
- Functions can take other functions as input and return other functions as output
    - These are called *higher-order functions*
- Functions are *pure*
    - They only act on their inputs
    - They are *referentially transparent*

Recap: Modularity

- Functions are building blocks and are treated like variables
- Functions can take other functions as input and return other functions as output
  - These are called *higher-order functions*
- Functions are *pure*
  - They only act on their inputs
  - They are *referentially transparent*
- Operations have *no side-effects*
  - Modifying "state" outside of a local environment
  - e.g. Setting a value in a globally cache

Recap: Modularity

- Functions are building blocks and are treated like variables
- Functions can take other functions as input and return other functions as output
  - These are called *higher-order functions*
- Functions are *pure*
  - They only act on their inputs
  - They are *referentially transparent*
- Operations have *no side-effects*
  - Modifying "state" outside of a local environment
  - e.g. Setting a value in a globally cache
- Perceived *immutability* of state
  - Constant "variables" only

Recap: Composition

Recap: Composition

- `map` transforms the element(s) of a structure
- `fold` (`reduce`) sequentially "folds" a traversable structure down into a new value

> Exercise 3

The execution order of a purely functional program is irrelevant. Why?

Monoids

"A monoid is a set that is closed under an associative binary operation and has an identity element I in S such that for all a in S, Ia=aI=a."

Monoids

A monoid is something we can `reduce`.

Monoids

A monoid is something we can `reduce`.

Let's take the set of integers: `[0, 1, 2..]`

```
2 + 0 = 2
```

Monoids

A monoid is something we can `reduce`.

+ is our associative binary operator

```
2 + 0 = 2
```

Monoids

A monoid is something we can `reduce`.

`0` is our identity element

```
2 + 0 = 2
```

Monoids

```
fold(+, 0)        // integers and addition
fold(*, 1)        // integers and multiplication
fold(+, [])       // arrays and concatenation
fold(+, "")       // strings and concatenation
fold(&&, true)    // booleans and logical AND
fold(||, false)   // booleans and logical OR
```

> Exercise 4

Write a *pure function* that takes a list of numbers, adds one to each number, and then returns the sum of the incremented numbers.

- Do not use using any loop constructs e.g `for`, `while`, etc
- Do not use `map` or `reduce`
- Do not use any built-in Javascript functions

🤔

Recursion

Recursion

Recursion is a way to "loop" over things

Recursion

Recursion is a way to "loop" over things

```
function loop(list) {
  if (list.length == 0) { return }
  console.log("loop")
  loop(list.slice(1))
}
```

Recursion

Recursion is when a function directly or indirectly calls itself. It can be used as a way to "loop" over things

```
function loop(list) {
  if (list.length == 0) { return }
  console.log("loop")
  loop(list.slice(1))
}
```

Recursion

Recursion is when a function directly or indirectly calls itself. It can be used as a way to "loop" over things

```
function loop(list) {
  if (list.length == 0) { return }
  console.log("loop")
  loop(list.slice(1))
}
```

Recursion

```
loop([1,2,3])
  console.log
  loop()
```

Recursion

```
loop([1,2,3])
  console.log
  loop([2, 3])
    console.log
    loop()
```

Recursion

```
loop([1,2,3])
  console.log
  loop([2, 3])
    console.log
    loop([3])
      console.log
      loop()
```

Recursion

```
loop([1,2,3])
  console.log
  loop([2, 3])
    console.log
    loop([3])
      console.log
      loop([])
        return
```

> Exercise 4

Write a *pure function* that takes a list of numbers, adds one to each number, and then returns the sum of the incremented numbers.

- Do not use using any loop constructs e.g `for`, `while`, etc
- Do not use `map` or `reduce`
- Do not use any built-in Javascript functions

> Exercise 4 Solution

```
function exercise4(list) {
  if (list.length == 0) {
    return 0
  } else {
    return (list[0] + 1) + exercise4(list.slice(1))
  }
}
```

> Exercise 5

Write a program that takes a list of numbers, adds one to each number, and then returns the sum of the incremented numbers.

- Do not use using any loop constructs e.g `for`, `while`, etc
- Do not use `map` or `reduce`
- Do not use any built-in Javascript functions

> Exercise 5

Write a program that takes a list of numbers, adds one to each number, and then returns the sum of the incremented numbers.

- Do not use using any loop constructs e.g `for`, `while`, etc
- Do not use `map` or `reduce`
- Do not use any built-in Javascript functions
- Do not use the + operator

> Exercise 5

Write a program that takes a list of numbers, adds one to each number, and then returns the sum of the incremented numbers.

- Do not use using any loop constructs e.g `for`, `while`, etc
- Do not use `map` or `reduce`
- Do not use any built-in Javascript functions
- Do not use the + operator
- Do not use numbers

> Exercise 5

Write a program that takes a list of numbers, adds one to each number, and then returns the sum of the incremented numbers.

- Do not use using any loop constructs e.g `for`, `while`, etc
- Do not use `map` or `reduce`
- Do not use any built-in Javascript functions
- Do not use the + operator
- Do not use numbers
- Do not use strings

> Exercise 5

Write a program that takes a list of numbers, adds one to each number, and then returns the sum of the incremented numbers.

- Do not use using any loop constructs e.g `for`, `while`, etc
- Do not use `map` or `reduce`
- Do not use any built-in Javascript functions
- Do not use the + operator
- Do not use numbers
- Do not use strings
- or built-in primitives or data structures

Peano Numbers

Peano Numbers

```
data Number = Zero | Succ Number
```

Peano Numbers

```
data Number = Zero | Succ Number
```

```
zero  = Zero
one   = Succ zero
two   = Succ Succ zero
three = Succ Succ Succ zero
```

Lambda Calculus

Lambda Calculus

In the 1930s, Alonzo Church published his papers on Lambda Calculus, a system for expressing any arbitrary computation or algorithm.

Lambda Calculus

In the 1930s, Alonzo Church published his papers on Lambda Calculus, a system for expressing any arbitrary computation or algorithm. In Lambda Calculus, you can only define a functions and call functions. Nothing more.

Lambda Calculus

Church encoded numbers in Lambda Calculus using something similar to Peano's numbers. These are called *Church Numerals*:

$0 = \lambda f.\lambda x.x$

$1 = \lambda f.\lambda x.fx$

$2 = \lambda f.\lambda x.ffx$

$3 = \lambda f.\lambda x.fffx$

Why are we talking about this?

Any problem that can be solved by a computer can be solved using functional programming

Please don't solve every problem using functional programming

Please don't solve every problem using functional programming

(please don't become a "functional programmer")

Where to go from here

Where to go from here

- Remember that FP provides tools for decomposing problems and *composing solutions*.
    - Composing solutions well is essential to managing complexity.

Where to go from here

- Remember that FP provides tools for decomposing problems and *composing solutions*.
  - Composing solutions well is essential to managing complexity.
- Start working in tiny bits of FP into your projects
  - Try refactoring a small function to be pure
  - Make things `const` whenever you can
  - Try using `map`, `reduce`, and `filter`

Where to go from here

- Remember that FP provides tools for decomposing problems and *composing solutions*.
  - Composing solutions well is essential to managing complexity.
- Start working in tiny bits of FP into your projects
  - Try refactoring a small function to be pure
  - Make things `const` whenever you can
  - Try using `map`, `reduce`, and `filter`
- Read "Why Functional Programming Matters" by John Hughes

Where to go from here

- Remember that FP provides tools for decomposing problems and *composing solutions*.
  - Composing solutions well is essential to managing complexity.
- Start working in tiny bits of FP into your projects
  - Try refactoring a small function to be pure
  - Make things `const` whenever you can
  - Try using `map`, `reduce`, and `filter`
- Read "Why Functional Programming Matters" by John Hughes
- Force yourself to program a little each week in a purely functional language!
  - Elm, Elixir, Haskell

Where NOT to go from here

Where NOT to go from here

- Don't rewrite everything in a functional style!
  - It will be very hard and you will be very sad

Where NOT to go from here

- Don't rewrite everything in a functional style!
  - It will be very hard and you will be very sad
- Don't read monad tutorials
  - Abstract concepts are hard to understand BECAUSE THEY ARE ABSTRACT. Get concrete knowledge first.

Where NOT to go from here

- Don't rewrite everything in a functional style!
  - It will be very hard and you will be very sad
- Don't read monad tutorials
  - Abstract concepts are hard to understand BECAUSE THEY ARE ABSTRACT. Get concrete knowledge first.
- Don't let jargon get you down

Where NOT to go from here

- Don't rewrite everything in a functional style!
  - It will be very hard and you will be very sad
- Don't read monad tutorials
  - Abstract concepts are hard to understand BECAUSE THEY ARE ABSTRACT. Get concrete knowledge first.
- Don't let jargon get you down
- Don't write inscrutable functional code that will make your co-workers and future self sad

Interested in more?

- Currying
- Partial Application
- Algebraic Data Types
- Tail recursion
- Functors
- Applicatives
- Monads
- "Point-free" style
- More common FP glue functions

Thanks! Questions?