Enjoy the Burrito

Christopher R. Genovese

Department of Statistics & Data Science

07 Nov 2024
Session #19

Plan

Infrastructure

Plan

Infrastructure

Case Study: Parsing

Plan

Infrastructure

Case Study: Parsing

Case Study: Algebraic Design

Announcements

- `fpc` code, python 3.12
- **Reading**: From last time:
  - Dynamic Programming
  - Category Theory for the Working Hacker
- **Homework**: `parser-combinators`, `kd-tree` Exercises #1 and #2

Plan

Infrastructure

Case Study: Parsing

Case Study: Algebraic Design

Review: Functional Programming

- Program as the composition of functions
  (composability, first-class functions)
- Expression oriented
- Clean separation of ***calculations*** and ***actions*** (pure functions)
- Referential transparency, equational reasoning
- Avoid mutating state (favor immutability except in limited scope)
- Declarative Structure
- Laziness, Closures, Higher-Order Functions, ...

Sums, Products, and Exponentials, Oh My

- `Void` :: The type with no realizable values; maps to 0.
- `Unit` :: The type with only one value; maps to 1.
- `Boolean` :: The type with two values (True and False): maps to 2.
- Product types :: Tuples and records, Cartesian product; maps to product

```
type Pair a b = Pair a b
type Triple a b c = Triple a b c
...
```

  Can write tuple types in literal form `(a, b)`, `(a, b, c)`, ...
  How many `(Boolean, Boolean)` tuples? How many `(Boolean, Boolean, Boolean)`?

- Sum types :: Alternatives, includes enums, **disjoint union**; maps to sum

```
type Boolean = False | True
type Color = Red | Green | Blue
type Maybe a = None | Some a
type Either a b = Left a | Right b
type These a b = This a | That b | Those a b
```

- Exponentials :: Functions; `a -> b` corresponds to b^a.
  How many functions of type `Boolean -> Pair Boolean Boolean`?

The Algebra of Types

Numbers	Types
0	`Void`
1	`Unit`
2	`Boolean`
$a + b$	`Either a b`
$a * b$	`Pair a b`
$1 + a$	`Maybe a`
b^a	`a -> b`

Use $\cong$ to denote isomorphism.

PUZZLE: Show that $2 \cong 1 + 1$. What does this mean in terms of the types above?

PUZZLE: Show that $a + a \cong 2 * a$. What does this mean?

PUZZLE: Show that $a * b \cong b * a$. What does this mean?

Recursive Types

```
type List a = Nil | Cons a (List a)
type BinaryTree a = Leaf | Branch (Tree a) a (Tree a)
type Tree a = Node a (List (Tree a))
```

We can translate these (loosely) into algebraic equations:

List a $x = 1 + ax$

$= 1 + a(1 + ax)$

$= 1 + a + a^2(1 + ax)$

$= \dots$

$= 1 + a + a * a + a * a * a + \cdots$

$= \frac{1}{1-a}$ $\qquad \frac{\mathrm{d}x}{\mathrm{d}a} = \frac{1}{(1-a)^2}$

BinaryTree a $x = 1 + ax^2$

$= 1 + a(1 + ax^2)^2$

$= 1 + a + 2a^2x^2 + a^3x^4 + \cdots$

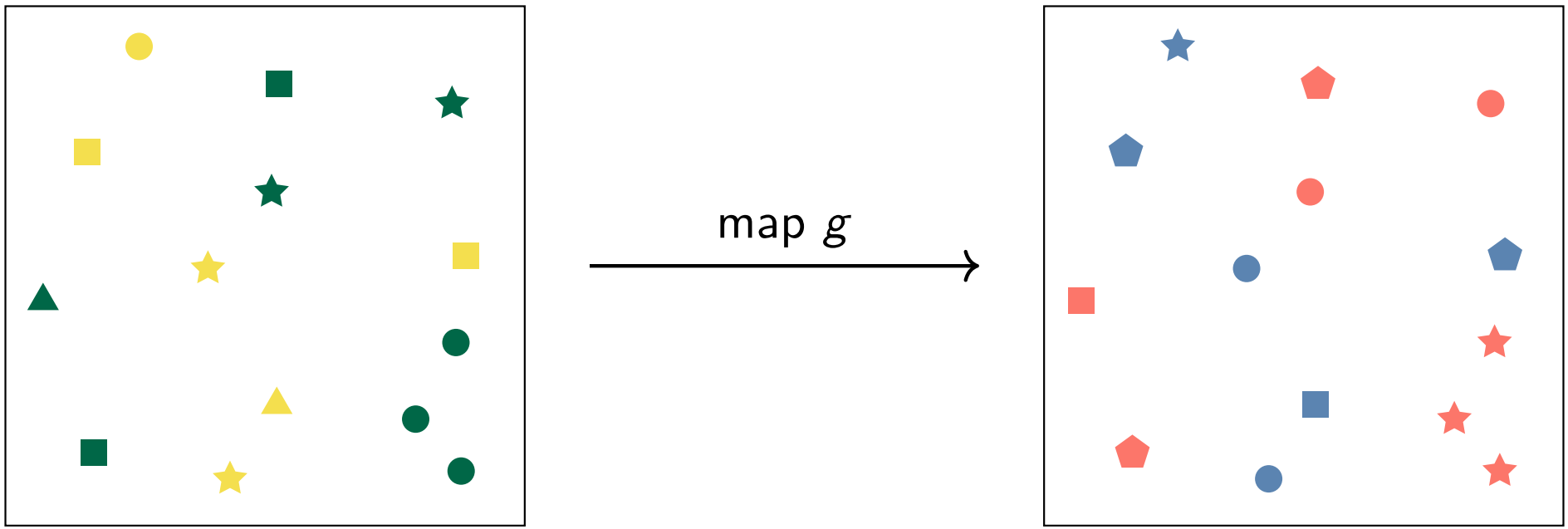
$= 1 + a + 2a^2 + 5a^3 + 14a^4 + 42a^5 + \cdots$

Tree a $x = a * (1 + x + x^2 + x^3 + \cdots) = \frac{a}{1-x} \implies x^2 - x + a = 0$ pushing it

$x = a + a^2 + 2a^3 + 5a^4 + 14a^5 + 42a^6 + 132a^7 + 429a^8 + \cdots$

Review: Functors

Computational context where we can transform the "results" inside it while preserving the context's "shape."




```
trait Functor (f : Type -> Type) where
    map : (a -> b) -> f a -> f b

laws Functor where
    map id == id
    map g . map h == map (g . h)
```

Review: Functors

What is the shape of a _____?

1. List
2. Pair
3. Dict
4. Maybe
5. Tree
6. Function `r -> __` (aka Reader r)
7. State s

FPC demos

Effects and Applicative Functors

Effects refer to ordinary computations/values augmented with some extra capabilities. The idea is quite general – and a bit vague – but useful.

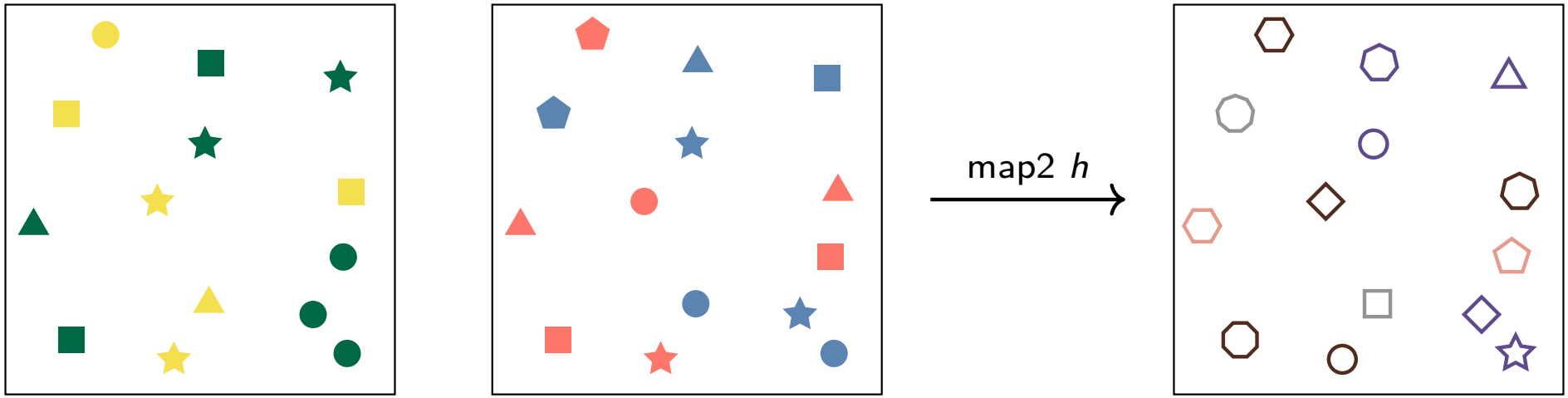
Examples:

- `Maybe a` describes an effect, the capability of *being possibly missing*.
- `Reader r` describes an effect, the capability of *having access to information in an environment*.
- `State s` describes an effect, the capability of *updating a state as part of computing a result*.
- *Side effects* are effects (but not necessarily vice-versa).

All of these can be expressed as Functors, but to make use of effects in practice, we need more power than vanilla Functors' map alone can give.

This leads to the idea of **Applicative Functors**.

Applicative Functors




```
trait Functor f => Applicative (f : Type -> Type) where
    pure : a -> f a
    map2 : (a -> b -> c) -> f a -> f b -> f c     -- lift2 := map2 h
    ap   : f (a -> b) -> f a -> f b

    unit : f Unit                                -- Unit equiv ()
    combine : f a -> f b -> f (a, b)
```

Can derive pairs pure and map2, pure and ap, and unit and combine from each other.

```
laws Applicative where
    combine unit a  ~= a ~= combine a unit
    combine a (combine b c) ~= combine (combine a b) c
    combine (map g fa) (map h fb) == bimap g h (combine fa fb)
```

Folds, Traversals, and Filters

Contexts that can be reduced to a summary value one piece at a time are ***foldable***:

```
trait Foldable (f : Type -> Type) where
    foldM : Monoid m => (a -> m) -> f a -> m
    fold  : (a -> b -> a) -> a -> f b -> a
```

Contexts in which elements can be removed are ***filterable***:

```
trait Functor f => Filterable (f : Type -> Type) where
    mapMaybe : (a -> Maybe a) -> f a -> f b
```

Contexts that can be transformed to one of the same *shape* by executing an effectful function one element at a time are ***traversable***:

```
trait (Functor t, Foldable t) => Traversable (t : Type -> Type) where
    traverse : Applicative f => (a -> f b) -> t a -> f (t b)
    sequence : Applicative f => t (f a) -> f (t a)
```

FPC Demos

Plan

Infrastructure

Case Study: Parsing

Case Study: Algebraic Design

What is a Parser?

A **parser** takes unstructured data as input and returns a structured representation of the data as output.

Classic examples:

- a programming language as text is converted to a *syntax tree* describing the code
- a text file in CSV format converted to a data frame,
- a series of network packets containing JSON converted to a nested record/dict.

Parsers and the operations that act on them to give *meaning* to the structured representations (e.g., compilers) have wide and frequent application.

Here, we will use parsers as a mechanism to think about programming approaches, patterns, and concepts.

We will build context-sensitive parsers out of smaller modular units called ***combinators***.

Starter Parsers

We want a parser that reads a string and gives us a character and a parser that reads a string and gives us a natural number. How can we represent these as types

Starter Parsers

We want a parser that reads a string and gives us a character and a parser that reads a string and gives us a natural number. How can we represent these as types

```
char : String -> Maybe Char
natural : String -> Maybe Natural
```

Why is Maybe here? What other basic operations might we want? Think about how we might process data or programs? How might we get two natural numbers in a CSV file? How might we read an alphabetic word? What operations do we need?

Examples: `splitAtChar`, `twoThings`

Can we compose parsers?

Digression: A Common Pattern

$$f(x) = 3x^2 + 4 = (\square + 4) \circ (3\square) \circ (\square^2)$$

Function composition is associative with a unit (a monoid).

We can think of programs as being composed in a similar way.

```
def a(x):
    print('Hello, ', end='')
    return x + 1

def b(x):
    print('world!')
    return x + 2

def main():
    c = a(1) + b(2)
    print( f'c = {c}')
    return 0
```

```
def alt_main():
    c = b(2) + a(1)
    print( f'c = {c}')
    return 0
```

Are `main` and `alt_main` the same program?

Digression: A Common Pattern (cont'd)

In Python/R, + is adding numbers, and addition should be *commutative*.

But we are not adding numbers, we are adding *programs*!

```
  a   : Int -> IO Int
  b   : Int -> IO Int
(+) : Int -> Int -> Int
```

We need a distinction between calculations and actions/effects/actions.

A Common Pattern (cont'd)

In general, we want to *compose* programs, but we cannot just do it (`Int -> IO Int` does not compose with `Int -> IO Int`).

```
   (.)     : (b ->    c) -> (a ->    b) -> (a ->    c)
semicolon : (b -> IO c) -> (a -> IO b) -> (a -> IO c)
```

Composition of programs – computations with context attached.

Examples of other computations:

- Async functions
- Random Variables
- Missing Data

Let's see these in action

Monads

A **monad** is a strategy for structuring, composing, and sequencing computations augmented with additional context.

```
trait Applicative m => Monad (m : Type -> Type) where
    bind : m a -> (a -> m b) -> m b
    join : m (m a) -> m a

    -- derived method, look familiar?
    kleisli : (b -> m c) -> (a -> m b) -> (a -> m c)
```

```
laws Monad where
  bind (pure x) f == f x
  bind m pure == m
  bind (bind m f) g == bind m (\x -> bind (f x) g)
```

Laws easier to express (and more familiar!) in terms of `kleisli`:

```
  kleisli pure f == f
  kleisli f pure == f
  kleisli (kleisli f g) h == kleisli f (kleisli g h)
```

Plan

Infrastructure

Case Study: Parsing

Case Study: Algebraic Design

Advent of Code Challenge: Part 1

You've managed to sneak in to the prototype suit manufacturing lab. The Elves are making decent progress, but are still struggling with the suit's size reduction capabilities.

While the very latest in 1518 alchemical technology might have solved their problem eventually, you can do better. You scan the chemical composition of the suit's material and discover that it is formed by extremely long polymers (one of which is available as your puzzle input).

The polymer is formed by smaller units which, when triggered, react with each other such that two adjacent units of the same type and opposite polarity are destroyed. Units' types are represented by letters; units' polarity is represented by capitalization. For instance, r and R are units with the same type but opposite polarity, whereas r and s are entirely different types and do not react.

For example:

- In `aA`, `a` and `A` react, leaving nothing behind.
- In `abBA`, `bB` destroys itself, leaving `aA`. As above, this then destroys itself, leaving nothing.
- In `abAB`, no two adjacent units are of the same type, and so nothing happens.
- In `aabAAB`, even though `aa` and `AA` are of the same type, their polarities match, and so nothing happens.

Now, consider a larger example, `dabAcCaCBAcCcaDA`.

```
dabAcCaCBAcCcaDA  The first 'cC' is removed.
dabAaCBAcCcaDA    This creates 'Aa', which is removed.
dabCBAcCcaDA      Either 'cC' or 'Cc' are removed (the result is the same).
dabCBAcaDA        No further actions can be taken.
```

After all possible reactions, the resulting polymer contains 10 units.

How many units remain after fully reacting the polymer you scanned?

Algebraic Structure

What are the data here? What is the algebraic structure?

```
represent : Data -> Structure
represent = foldM inject

inject : Component -> Structure
inject c
    | isAlpha c and isLowerCase c =
    | isAlpha c and isUpperCase c =
    | otherwise                   = munit
```

Advent of Code Challenge: Part 2

Time to improve the polymer.

One of the unit types is causing problems; it's preventing the polymer from collapsing as much as it should. Your goal is to figure out which unit type is causing the most problems, remove all instances of it (regardless of polarity), fully react the remaining polymer, and measure its length.

For example, again using the polymer `dabAcCaCBAcCcaDA` from above:

- Removing all A/a units produces `dbcCCBcCcD`. Fully reacting this polymer produces `dbCBcD`, which has length 6.
- Removing all B/b units produces `daAcCaCAcCcaDA`. Fully reacting this polymer produces `daCAcaDA`, which has length 8.
- Removing all C/c units produces `dabAaBAaDA`. Fully reacting this polymer produces `daDA`, which has length 4.
- Removing all D/d units produces `abAcCaCBAcCcaA`. Fully reacting this polymer produces `abCBAc`, which has length 6.

In this example, removing all `C`/`c` units was best, producing the answer 4.

What is the length of the shortest polymer you can produce by removing all units of exactly one type and fully reacting the result?

Algebraic Structure

Here, we are *mapping* between structures.

The End