Make Your Own Burrito

Christopher R. Genovese

Department of Statistics & Data Science

12 Nov 2024
Session #20

Plan

Algebraic Design

Plan

Algebraic Design

Case Study: Building Parsers

Plan

Algebraic Design

Case Study: Building Parsers

More Power in Composition

Plan

Algebraic Design

Case Study: Building Parsers

More Power in Composition

Case Study: A Simple Algebraic Design

Plan

Algebraic Design

Case Study: Building Parsers

More Power in Composition

Case Study: A Simple Algebraic Design

Appendix

Announcements

- `fp-concepts` package, get from Documents
- **Reading**: From last time:
  - Deriving the Z-combinator and Classes with the Z Combinator
  - Category Theory for the Working Hacker
- **Homework**: `parser-combinators` (new exercises), `kd-tree` Exercises #1 and #2

Plan

Algebraic Design

Case Study: Building Parsers

More Power in Composition

Case Study: A Simple Algebraic Design

Appendix

Functional and Algebraic Design

Our goal is to design **larger abstractions that compose**.

We realize this through the **composition of algebras**.

We want to:

- Separate the *what* from the *how*. (Declarative structure)
- Separate *calculations* from *effects*. (Referential Transparency)
- Separate "denotational" from "operational" semantics. (Meaning, Programming to the Interface)

Denotational Semantics

> Think of the programs and objects they manipulate as realizations of abstract mathematical objects.
>
> Express meaning of the program through these mathematical relationships.
>
> This mapping is independent of implementation.

Operational Semantics

> Formalizes the low-level steps of computation
>
> "One must mentally execute it to understand it." – John Backus

Components of an Algebra

Illustrate with the `Maybe a` type.

Components of an Algebra

Illustrate with the `Maybe a` type.

1. Carrier Type: Represents the target result type of the computation.
   `a`

Components of an Algebra

Illustrate with the `Maybe a` type.

1. Carrier Type: Represents the target result type of the computation.
2. **Introduction Forms**: Creates elements of the algebra.
   `Some`, `None_` (or equivalently `pure`, `empty`)

Components of an Algebra

Illustrate with the `Maybe a` type.

1. Carrier Type: Represents the target result type of the computation.
2. **Introduction Forms**: Creates elements of the algebra.
3. **Combinators**: Operations to transform and manipulate objects in the algebra.

   `map`, `map2`, `ap`, `alt`, `bind`, ..., e.g.,

   ```
   ap(pair, Some(4), Some(10))
   ap(pair, Some(4), None_())

   ap(triple, pure(4), pure(10), pure(12))
   ap(triple, pure(4), empty, pure(12))
   ```

Components of an Algebra

Illustrate with the `Maybe a` type.

1. Carrier Type: Represents the target result type of the computation.
2. **Introduction Forms**: Creates elements of the algebra.
3. **Combinators**: Operations to transform and manipulate objects in the algebra.
4. **Elimination Forms**: Extracts results from the algebra

```
maybe(z, f, m)         # map None_() to z, Some(a) to f(a)
maybe(z, identity, m)  # == m.get(default)
```

Components of an Algebra

Illustrate with the `Maybe a` type.

1. Carrier Type: Represents the target result type of the computation.
2. **Introduction Forms**: Creates elements of the algebra.
3. **Combinators**: Operations to transform and manipulate objects in the algebra.
4. **Elimination Forms**: Extracts results from the algebra
5. **Laws**: Properties/constraints that govern the operations

```
map(f, None_()) == None_()
bind(f, None_()) == None_()

alt(None_(), x) == x
alt(x, None_()) == x
alt(x, alt(y, z)) == alt(alt(x, y), z)
```

Components of an Algebra

Illustrate with the `Maybe a` type.

1. Carrier Type: Represents the target result type of the computation.
2. **Introduction Forms**: Creates elements of the algebra.
3. **Combinators**: Operations to transform and manipulate objects in the algebra.
4. **Elimination Forms**: Extracts results from the algebra
5. **Laws**: Properties/constraints that govern the operations

This applies broadly, and we will build these for `Parser`

Essential Features of an Algebra

1. Composable
2. Lawful
3. Polymorphic
4. Uses the Least Powerful Abstraction that works
5. Specifies an interface/contract through operations and laws
6. Open to multiple interpretations/implementations

Reminder: Effects

Effects refer to ordinary computations/values augmented with some extra capabilities. We represent effects with types.

f a

There are many different, commonly-used effects.

Reminder: Effects

Effects refer to ordinary computations/values augmented with some extra capabilities. We represent effects with types.

f a

the effect context

results of the computation

Reminder: Effects

`List a` (non-determinism)

`Pair c a` (conjunction)

`Maybe a` (partiality)

`Either e a` (disjunction)

`IO a` (input/output)

`Reader r a` (environment)

`...` many more

`Random g a` (randomness)

`Writer w a` (logging)

`State s a` (updating state)

Plan

Algebraic Design

Case Study: Building Parsers

More Power in Composition

Case Study: A Simple Algebraic Design

Appendix

Recap

Last time, we started building an algebra for parsers.

We started with a parser `char` to (maybe) read a character from a string, of type `String -> Maybe Char`. But this left us unable to do anything else.

Recap

Last time, we started building an algebra for parsers.

We extended this to type `String -> Pair String (Maybe Char)` and similarly for the parser `natural` of type `String -> Pair String (Maybe Natural)`.

This suggests a type

```
type Parser a = String -> Pair String (Maybe a)
```

Recap

Last time, we started building an algebra for parsers.

This suggests a type

```
type Parser a = String -> Pair String (Maybe a)
```

We worked a bit to compose these parsers.

```
both : Parser a -> Parser b -> Parser (Pair a b)

def both(parser1: Parser[a], parser2: Parser[b]) -> Parser[Pair[a, b]]:
    def do_both(input1: str) -> Pair[str, Maybe[Pair[a, b]]]:
        input2, mc = parser1(input1)
        if not mc:
            return (input, None_())
        input3, mn = parser2(input2)
        if not mn:
            return (input, None_())
        return (input3, map2(pair, mc, mn))
    return do_both
```

We will come back to this idea in a bit

Plan

We will build up to a fully functional parser algebra, starting from simple parsers, and adding power to the abstraction as we go.

Things to look for:

Components

1. Carrier type
2. Introduction Forms
3. Combinators
4. Elimination Forms
5. Laws

Features:

1. Composable
2. Lawful
3. Polymorphic
4. Uses the Least Powerful Abstraction that works
5. Specifies an interface/contract
6. Open

Plan (cont'd)

Let's begin with a couple more starter parsers to sharpen the ideas and we'll build from there.

```
void : Parser Unit                          -- Parser that always fails
empty : Parser Unit                         -- Parser that always succeeds
char : Char -> Parser Char                  -- Parser that accepts only given
chars_in : Iterable Char -> Parser Char     -- Parser that accepts one of give
natural : Parser Natural                    -- Parser for natural numbers
```

And a few combinators (among many)

```
use : Parser a -> b -> Parser B   - Uses value when given parser succeeds
alt : Parser a -> Parser b -> Parser (Either a b)
seq : Parser a -> Parser b -> Parser (Pair a b)
optional: Parser a -> a -> Parser a
```

Building . . .

Plan

Algebraic Design

Case Study: Building Parsers

More Power in Composition

Case Study: A Simple Algebraic Design

Appendix

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.

```python
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
```

```python
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

Algebraic Design

Case Study: Building Parsers

More Power in Composition

Case Study: A Simple Algebraic Design

Appendix

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?

Part 1

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?

Part 2

Here, we are *mapping* between algebraic structures.

What is the mapping?

Plan

Algebraic Design

Case Study: Building Parsers

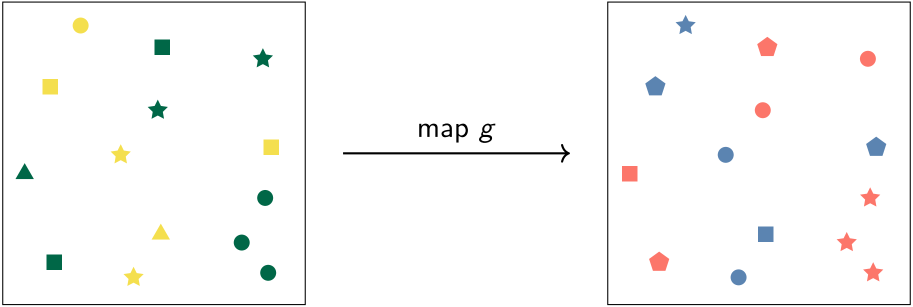
More Power in Composition

Case Study: A Simple Algebraic Design

Appendix

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.

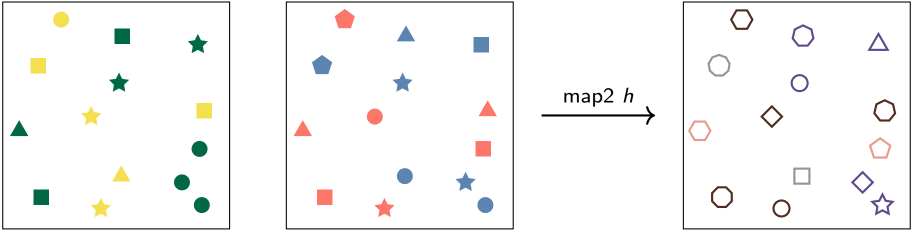
Examples:

- `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

The End