Programming with Algebraic Structures

Haskell Edition

Susan Potter

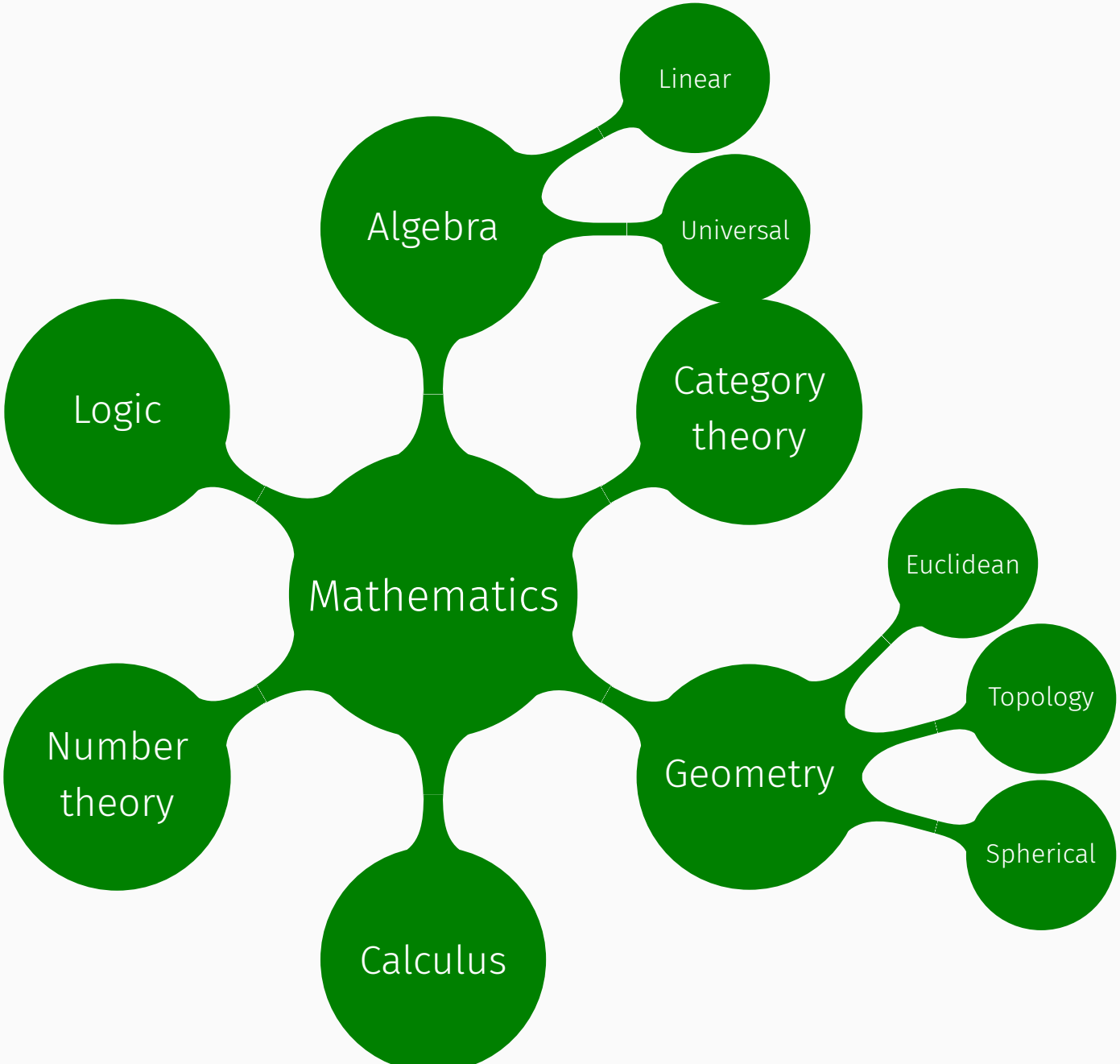
2019-10-27

Mathematics

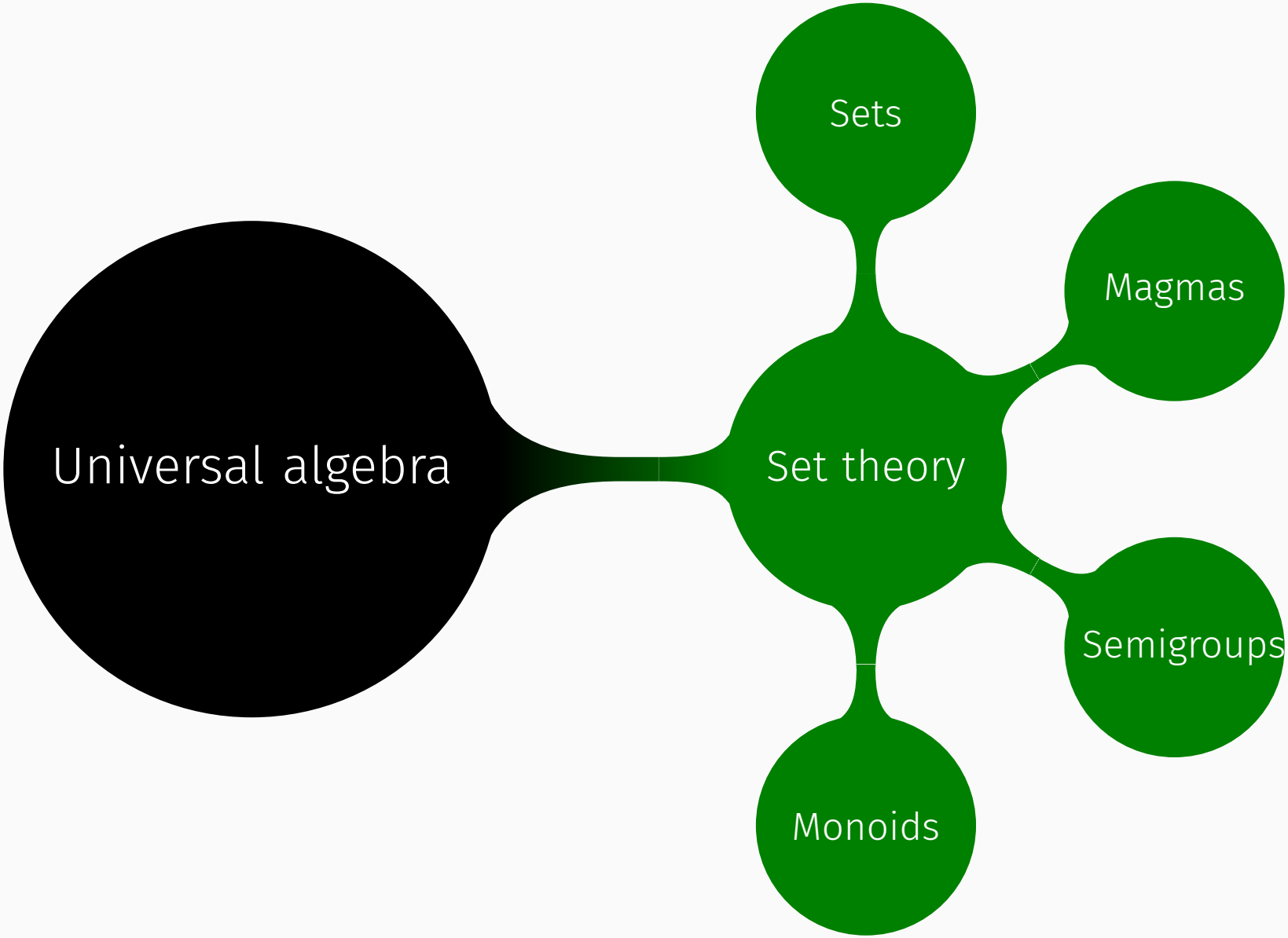
Why Math(s)?

I am a programmer/developer/software "engineer", why would I care about mathematics?
– You?

What kind of mathematics? (a super tiny view of the mindmap)



Today: how to apply Set theoretic abstractions



[Set-based] Algebraic Structures

What is a Set? (in Set theory)

> *A set in mathematics is a collection of well defined and distinct objects, considered as an object in its own right.*
> *– Wikipedia (c2018)*

- officially introduced in 1874 in a paper from Georg Cantor.
- $\{apple, banana, cantelope, durian\}$ is set of fruits I will eat.
- $\{1, 2, 3, -2, 2\}$ is not a set; it has duplicate element 2.
- $\{1, 2, 3\} \subset \mathbb{N}$ (read: the set consisting of 1, 2, and 3 is a subset of the natural numbers)

What is a Set? (in Haskell)

In Term/Value Space

```haskell
data Set a
  = MkSet (a -> Bool)

evens :: Set Int
evens = MkSet $
  \x -> x % 2 == 0

odds = MkSet $
  \x -> x % 2 == 1
```

In Type Definition Space

```haskell
data Even
  = EvenZero
  | EvenSucc Odd

data Odd
  = OddSucc Even
```

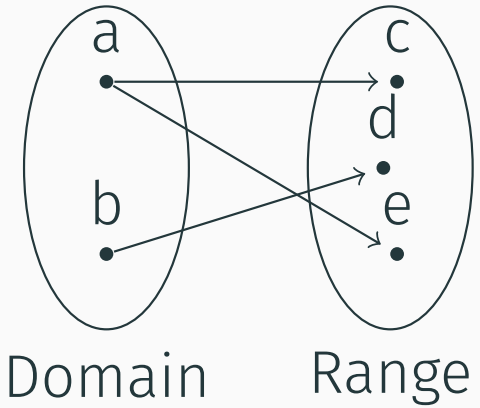
What is a Relation? (in Set theory)

A relation is a mapping of values between

- a domain (input) set; and
- a range (output) set

Note: The range (or output) set is sometimes called codomain.

Diagram



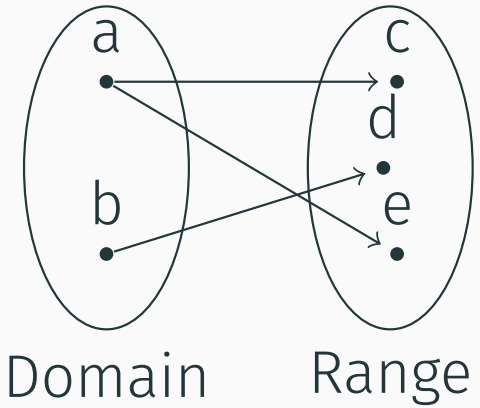
What is a Relation? (in Set theory)

A relation is a mapping of values between

- a domain (input) set; and
- a range (output) set

Note: The range (or output) set is sometimes called codomain.

Diagram



Table

Domain		Range
a	$\mapsto$	c
a	$\mapsto$	e
b	$\mapsto$	d

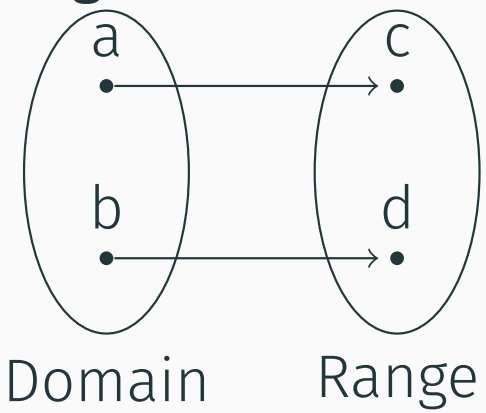
How to encode relations with functions? (in Haskell)

What is a Function? (in Set theory)

A function is a relation between a domain and range that associates every element of the domain to exactly one of the elements in the range.

The relation previously is not a function but is this?

Diagram

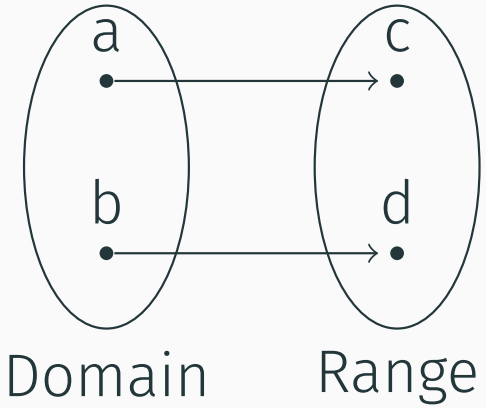


What is a Function? (in Set theory)

A function is a relation between a domain and range that associates every element of the domain to exactly one of the elements in the range.

The relation previously is not a function but is this?

Diagram



Table

Domain		Range
a	$\mapsto$	c
b	$\mapsto$	d

Is a Haskell function always a mathematical (Set theory) theory function?

Consider this Haskell function?

```haskell
-- Undefined for y=0
div
  :: Int -- ^ numerator
  -> Int -- ^ denominator
  -> Int -- ^ result
div x y = x / y
```

Is a Haskell function always a mathematical (Set theory) theory function?

Consider this Haskell function?

```haskell
-- Undefined for y=0
div
  :: Int -- ^ numerator
  -> Int -- ^ denominator
  -> Int -- ^ result
div x y = x / y
```

Is this a ...

- relation: Yes
- function: No

What is an algebra? (in Set theory)

- a set S (called the carrier)
- zero or more operations
  - which are functions $S^k -> S$
  - where k is the number of arguments (aka arity)
  - examples of operations:
    - unary (one argument) such as negation
    - binary (two arguments) such as addition

What is an algebra? (in Haskell)

- an algebraic data type `t` (you can call it carrier)
- zero or more operations over the type `t`
  - which are Haskell functions
  - examples include:

    `neg : t -> t` (unary operator)

    `(+) : Num t => t -> t -> t` (binary operator)

Why do we care?

- Offers abstraction allowing us to reason about visible common structure over many different types.
- We can encapsulate internal representation easily.
- While we write generic, reusable code.

What is an axiom?

An initial fact that does not need to be proved and always considered true.

From Greek root axioma which means, *"that which is thought worthy or fit."*

Common axioms for algebras?

Axiom name	Axiom property
Associativity (assoc)	`x + (y + z) = (x + y) + z`
Identity (id)	`0 + x = x + 0 = x`
Inverse (inv)	`x + (-x) = (-x) + x = 0`
Commutativity (comm)	`x + y = y + x`
Distributivity (dist)	`x*(y + z) = x*y + x*z`

Kinds of Algebras (Math)

Name	Binary	Unary	Axioms
Set			
Magma	`+`		
Semigroup	`+`		+(assoc)
Monoid	`+`		+(assoc, id)
Group	`+`	`neg`	+(assoc, id, inv)
Abelian Group	`+`	`neg`	+(assoc, id, inv, comm)
Rng	`+, *`		+(assoc, id, inv, comm, dist) *(assoc)
Ring	`+, *`		+(assoc, id, inv, comm, dist) *(assoc, id)
Field	`+, *`	`inv`	+(assoc, id, inv, comm, dist) *(assoc, id, inv)

Note: `+` and `*` are more general than numeric addition or multiplication above.

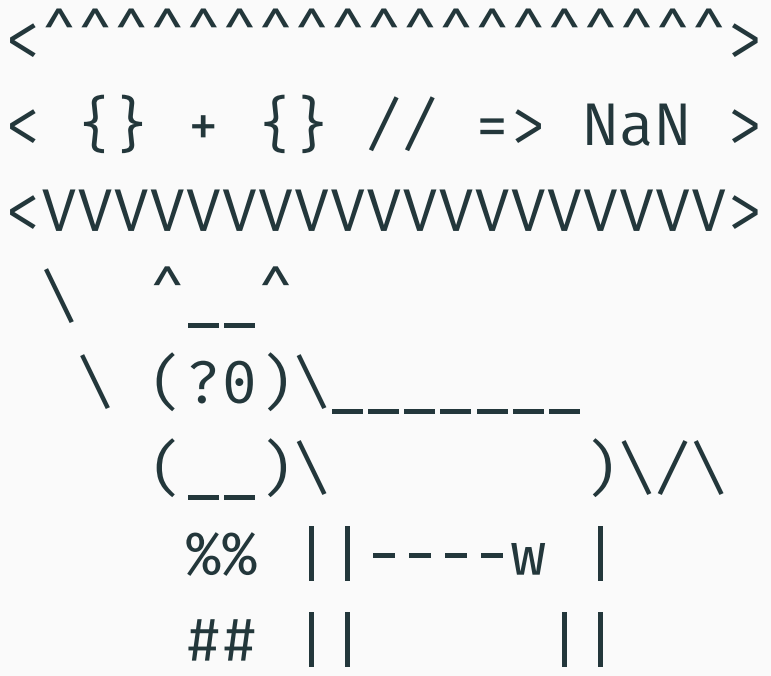
Application => Approximation

Math

Application => Approximation

Math

Programming



Apparently in Javascript: `{} + {} = NaN`

Examples of Algebras Programmers Care About?

Algebra kind	Example
Set	data type w/ zero closed functions
Magma	data type w/ one closed binary operator
Semigroup	appending non-empty lists
Monoid	Redis commands can be pipelined
Group	integers with +, neg, 0.

Set (Type): An Example

Car: An algebra over a product type carrier

```
data Car
  = MkCar
  { carMake  :: Make
  , carModel :: Model
  , carYear  :: Year
  }
```

- `Car` is a product type which is the carrier
- Zero operations closed over itself
- Elements of this set are values of this type

Beyond Sets

Role: An algebra over a sum type (enum) carrier

```haskell
data Role = Admin | User | Anon

-- Given two Roles produce the least
-- privileged role of the two
leastPrivilege :: Role -> Role -> Role
```

- `Role` is a sum type which is the carrier
- One binary operation closed over itself
- Is `leastPrivilege` operation associative?
- Is `leastPrivilege` operation commutative?
- Does this algebra have an identity element?
- Let's explore!

Role: `leastPrivilege` definition

```
leastPrivilege :: Role -> Role -> Role
leastPrivilege Anon     _        = Anon
leastPrivilege _        Anon     = Anon
leastPrivilege User     _        = User
leastPrivilege _        User     = User
leastPrivilege _        _        = Admin
```

Role: Is `leastPrivilege` operation associative?

```
genRole = element [ Admin, User, Anon ]
forAllRoles = forAll getRole

propLeastPrivilegeAssoc =
  property $ do
    x <- forAllRoles
    y <- forAllRoles
    z <- forAllRoles
    rAssoc === lAssoc
  where add    = leastPrivilege
        rAssoc = x `add` (y `add` z)
        lAssoc = (x `add` y) `add` z
```