Type

All dynamic languages are static languages with a single type

Any

Any => Any

```
def inc(value: Any): Any = value match {
  case x: Int    => x + 1
  case x: Double => x + 1
  case x: Char   => x.toString + "1"
  case x: String => x + "1"
}
```

Any => Any

```
def inc(value: Any): Any = value match {
  case x: Int    => x + 1
  case x: Double => x + 1
  case x: Char   => x.toString + "1"
  case x: String => x + "1"
}
```

```
scala> inc(5)
res0: Any = 6

scala> inc(10.3)
res1: Any = 11.3

scala> inc('c')
res2: Any = c1
```

Any => Any

```
def inc(value: Any): Any = value match {
  case x: Int    => x + 1
  case x: Double => x + 1
  case x: Char   => x.toString + "1"
  case x: String => x + "1"
}
```

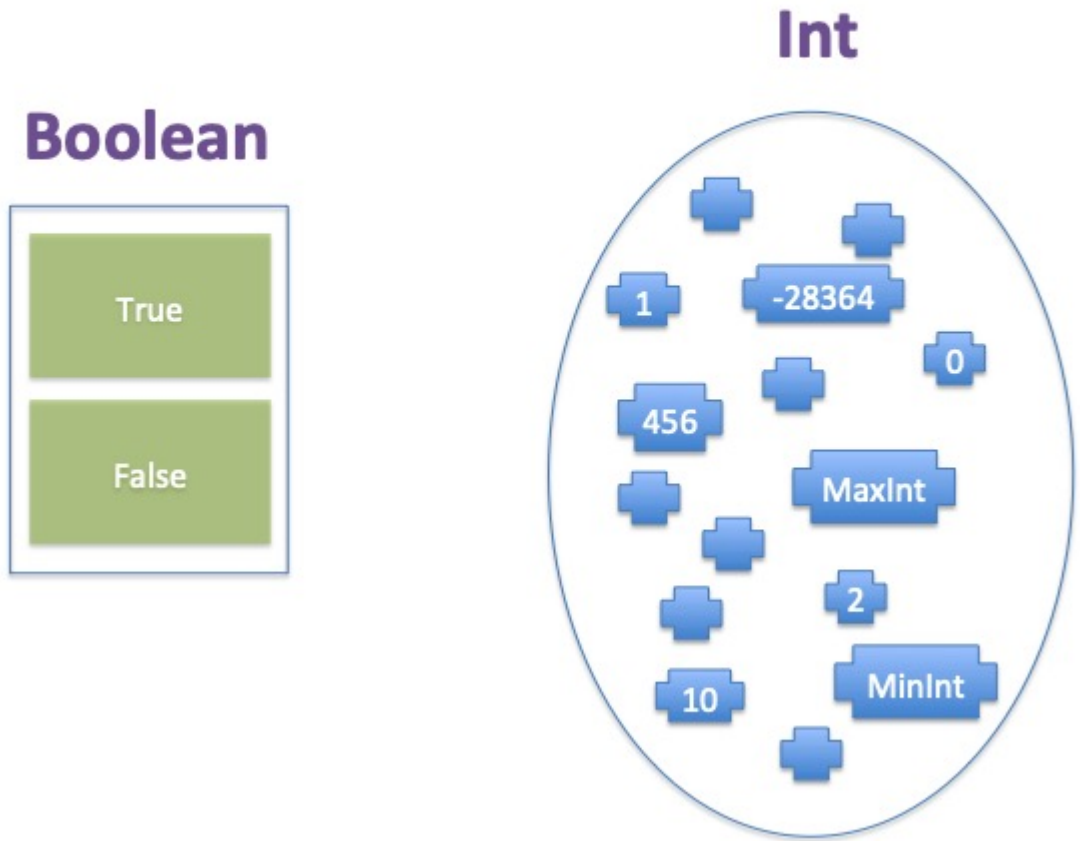
```
scala> inc(5)
res0: Any = 6

scala> inc(10.3)
res1: Any = 11.3

scala> inc('c')
res2: Any = c1
```

```
scala> inc(java.time.Instant.ofEpochMilli(0))
scala.MatchError: 1970-01-01T00:00:00Z (of class java.time.Instant)
  at .inc(<console>:12)
  ... 43 elided
```

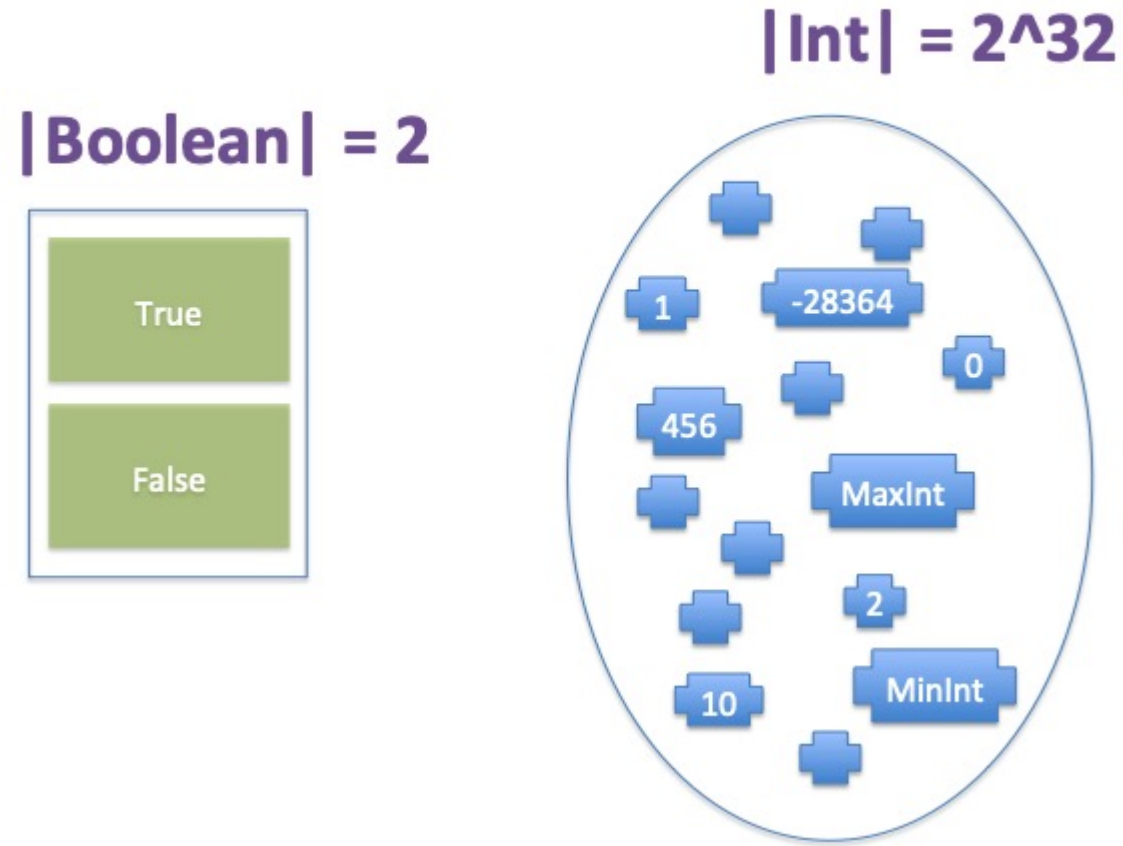
Plan

- What is the impact of types on our program? How to measure it?
- How to select types and tests to write more correct programs
- Explore relationship between types, algebra and logic

Type



Type



Cardinality of a type should exactly fit business requirement

Type too big

```
def getCurrency(country: String): String =
  country match {
    case "France" | "Germany" => "EUR"
    case "United Kingdom"     => "GBP"
    case "Switzerland"        => "CHF"
    case _                    => ???
  }
```

Type too big

```
def getCurrency(country: String): Option[String] =
  country match {
    case "France" | "Germany" => Some("EUR")
    case "United Kingdom"     => Some("GBP")
    case "Switzerland"        => Some("CHF")
    case _                    => None
  }
```

Type just right

```
sealed trait Country

case object France        extends Country
case object Germany       extends Country
case object UnitedKingdom extends Country
case object Switzerland   extends Country

sealed trait Currency

case object EUR extends Currency
case object GBP extends Currency
case object CHF extends Currency

def getCurrency(country: Country): Currency =
  country match {
    case France | Germany => EUR
    case UnitedKingdom    => GBP
    case Switzerland      => CHF
  }
```

Type too big

```
case class Person(name: String, streetNumber: Option[Int], streetName: Option[String])

def fullAddress(person: Person): Option[String] =
  (person.streetNumber, person.streetName) match {
    case (Some(x), Some(y)) => Some(s"$x $y")
    case (Some(x), None   ) => ???
    case (None   , Some(y)) => ???
    case (None   , None   ) => None
  }
```

```
scala> fullAddress(Person("John", Some(108), Some("Canon Street")))
res4: Option[String] = Some(108 Canon Street)

scala> fullAddress(Person("John", None, None))
res5: Option[String] = None
```

Type just right

```
case class Address(streetNumber: Int, streetName: String)
case class Person(name: String, address: Option[Address])

def fullAddress(address: Address): String =
  s"${address.streetNumber} ${address.streetName}"

def fullPersonAddress(person: Person): Option[String] =
  person.address.map(fullAddress)
```

Type too big

```
// quantity must be positive
case class Item(id: String, quantity: Int, price: Double)

// order must have at least one item
case class Order(id: String, items: List[Item]){
  def total: Double =
    items.map(i => i.quantity * i.price).sum
}
```

Type too big

```
// quantity must be positive
case class Item(id: String, quantity: Int, price: Double)

// order must have at least one item
case class Order(id: String, items: List[Item]){
  def total: Double =
    items.map(i => i.quantity * i.price).sum
}
```

```
def halfPriceFirstItem(order: Order): Order = {
  val firstItem = order.items.head
  val discounted = firstItem.copy(price = firstItem.price / 2)
  order.copy(items = discounted :: order.items.tail)
}
```

Type too big

```
// quantity must be positive
case class Item(id: String, quantity: Int, price: Double)

// order must have at least one item
case class Order(id: String, items: List[Item]){
  def total: Double =
    items.map(i => i.quantity * i.price).sum
}
```

```
def halfPriceFirstItem(order: Order): Order = {
  order.items match {
    case Nil     => ???
    case x :: xs =>
      val discounted = x.copy(price = x.price / 2)
      order.copy(items = discounted :: xs)
  }
}
```

Type just right

```
import cats.data.NonEmptyList
import cats.implicits._

// quantity must be positive
case class Item(id: String, quantity: Int, price: Double)

case class Order(id: String, items: NonEmptyList[Item]){
  def total: Double =
    items.foldMap(i => i.quantity * i.price)
}
```

```
def halfPriceFirstItem(order: Order): Order = {
  val firstItem  = order.items.head
  val discounted = firstItem.copy(price = firstItem.price / 2)
  order.copy(items = NonEmptyList(discounted, order.items.tail))
}
```

Even further

```
import cats.data.NonEmptyList
import cats.implicits._
import eu.timepit.refined.types.numeric.PosInt

case class Item(id: String, quantity: PosInt, price: Double)

case class Order(id: String, items: NonEmptyList[Item]){
  def total: Double =
    items.foldMap(i => i.quantity.value * i.price)
}
```

```
def halfPriceFirstItem(order: Order): Order = {
  val firstItem  = order.items.head
  val discounted = firstItem.copy(price = firstItem.price / 2)
  order.copy(items = NonEmptyList(discounted, order.items.tail))
}
```

Type too small

```
import java.util.Date
```

```
scala> val fifthMarch = new Date(1551818168101L)
fifthMarch: java.util.Date = Tue Mar 05 20:36:08 UTC 2019
```

Type too small

```
import java.util.Date
```

```
scala> val fifthMarch = new Date(1551818168101L)
fifthMarch: java.util.Date = Tue Mar 05 20:36:08 UTC 2019
```

```
scala> List(1,2,3,4).size
res11: Int = 4
```

Type too small

```
import java.util.Date
```

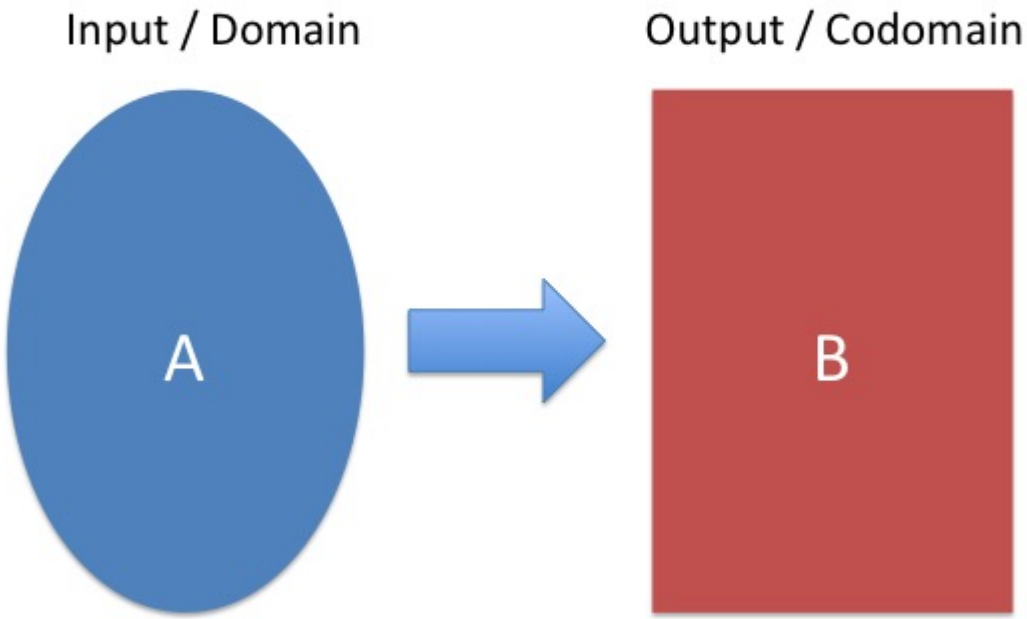
```
scala> val fifthMarch = new Date(1551818168101L)
fifthMarch: java.util.Date = Tue Mar 05 20:36:08 UTC 2019
```

```
scala> List(1,2,3,4).size
res11: Int = 4
```

```
def parseJson(json: String): Option[Json] = ???
```

Function

Function



```
val isEven: Int => Boolean =
      x => x % 2 == 0

val increment: Int => Int =
        x => x + 1
```

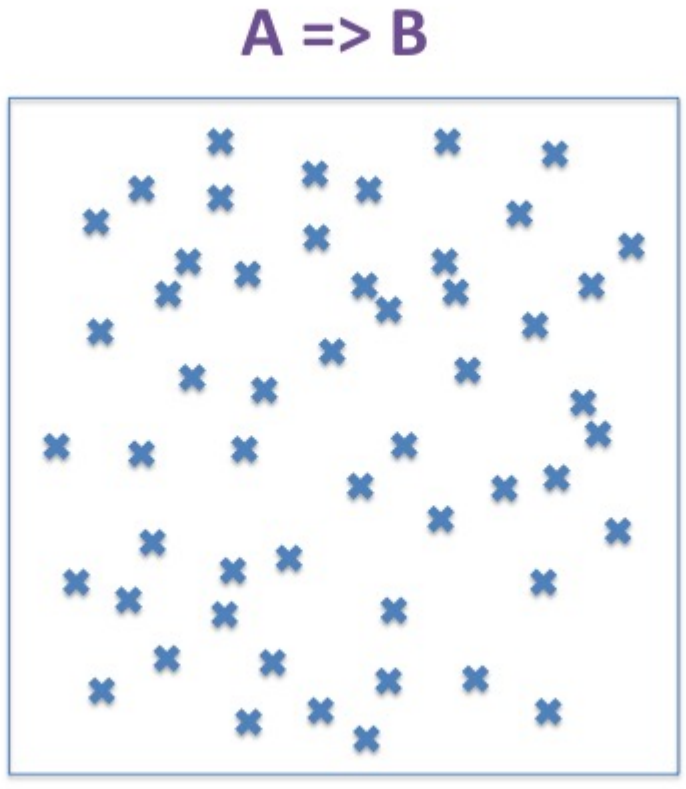
Type is a set

Type is a set

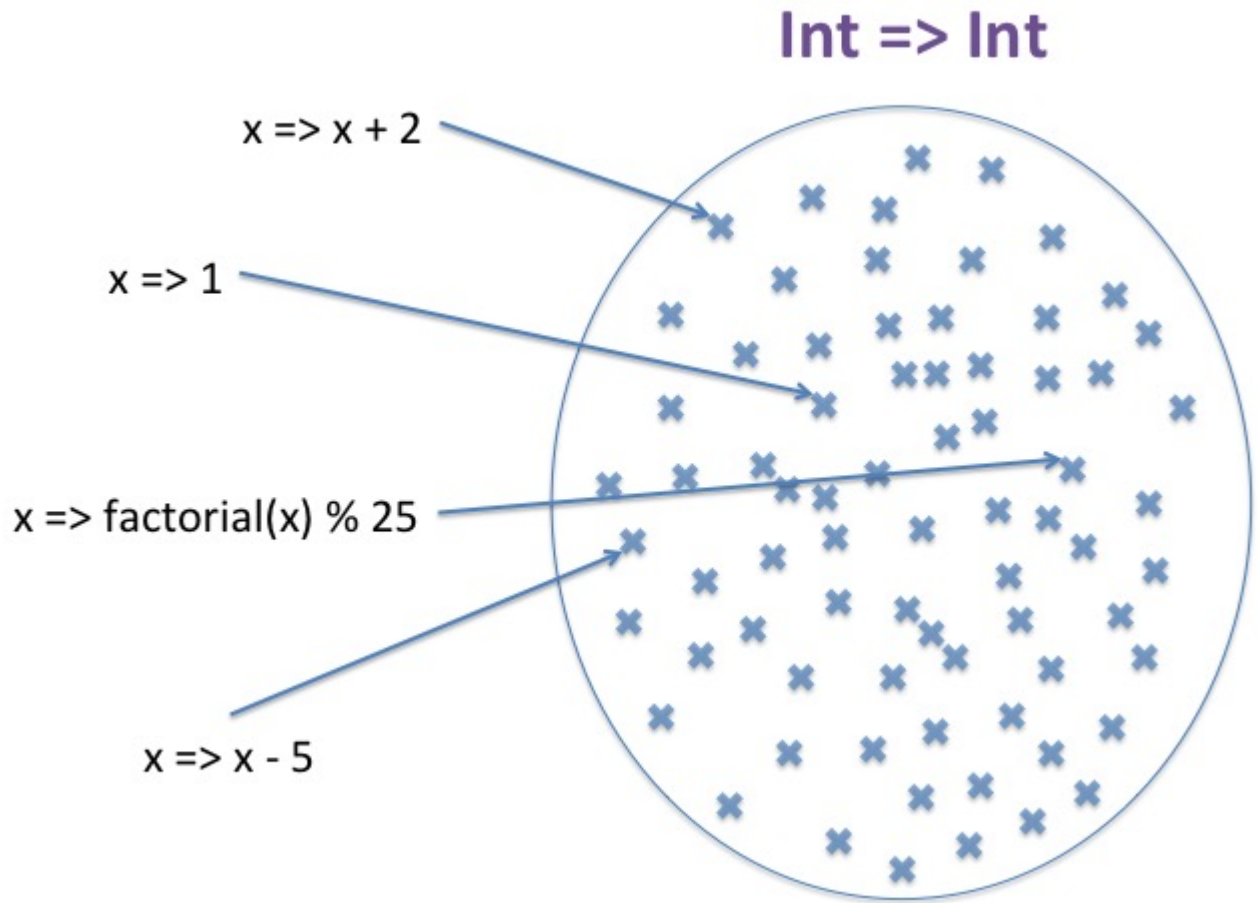
&&

A => B is a type

A => B is a set!



Function is a set!



|A => B| represents how many implementations exist

The smaller |A => B|, the better

Ultimate goal |A => B| = 1

How to calculate |A => B| ?

```
def getCurrency1(country: String ): Option[String] = ???

def getCurrency2(country: Country): Currency = ???
```

How to calculate |A => B| ?

```
def getCurrency1(country: String ): Option[String] = ???

def getCurrency2(country: Country): Currency = ???
```

Intuitively

```
|getCurrency2| < |getCurrency1|
```

Exercise 1

Sealed trait cardinality

```
sealed trait IntOrBoolean

case class AnInt(value: Int) extends IntOrBoolean
case class ABoolean(value: Boolean) extends IntOrBoolean
```

Sealed trait cardinality

```
sealed trait IntOrBoolean

case class AnInt(value: Int) extends IntOrBoolean
case class ABoolean(value: Boolean) extends IntOrBoolean
```

```
AnInt(Int.MinValue) // ~ -2 billion
...
AnInt(0)
AnInt(1)
...
AnInt(Int.MaxValue) // ~ +2 billion
```

Sealed trait cardinality

```
sealed trait IntOrBoolean

case class AnInt(value: Int) extends IntOrBoolean
case class ABoolean(value: Boolean) extends IntOrBoolean
```

```
AnInt(Int.MinValue) // ~ -2 billion
...
AnInt(0)
AnInt(1)
...
AnInt(Int.MaxValue) // ~ +2 billion
```

```
ABoolean(false)
ABoolean(true)
```

Sealed trait cardinality

```
sealed trait IntOrBoolean

case class AnInt(value: Int) extends IntOrBoolean
case class ABoolean(value: Boolean) extends IntOrBoolean
```

```
AnInt(Int.MinValue) // ~ -2 billion
...
AnInt(0)
AnInt(1)
...
AnInt(Int.MaxValue) // ~ +2 billion
```

```
ABoolean(false)
ABoolean(true)
```

```
|IntOrBoolean| = |AnInt| + |ABoolean|
                = |Int|   + |Boolean|
```

Case class cardinality

```
case class IntAndBoolean(i: Int, b: Boolean)
```

Case class cardinality

```
case class IntAndBoolean(i: Int, b: Boolean)
```

```
IntAndBoolean(Int.MinValue, false)  // ~ -2 billion
...
IntAndBoolean(0, false)
IntAndBoolean(1, false)
...
IntAndBoolean(Int.MaxValue, false) // ~ +2 billion
```

Case class cardinality

```
case class IntAndBoolean(i: Int, b: Boolean)
```

```
IntAndBoolean(Int.MinValue, false)  // ~ -2 billion
...
IntAndBoolean(0, false)
IntAndBoolean(1, false)
...
IntAndBoolean(Int.MaxValue, false) // ~ +2 billion
```

```
IntAndBoolean(Int.MinValue, true)  // ~ -2 billion
...
IntAndBoolean(0, true)
IntAndBoolean(1, true)
...
IntAndBoolean(Int.MaxValue, true) // ~ +2 billion
```

Case class cardinality

```
case class IntAndBoolean(i: Int, b: Boolean)
```

```
IntAndBoolean(Int.MinValue, false)  // ~ -2 billion
...
IntAndBoolean(0, false)
IntAndBoolean(1, false)
...
IntAndBoolean(Int.MaxValue, false) // ~ +2 billion
```

```
IntAndBoolean(Int.MinValue, true)  // ~ -2 billion
...
IntAndBoolean(0, true)
IntAndBoolean(1, true)
...
IntAndBoolean(Int.MaxValue, true) // ~ +2 billion
```

```
|IntAndBoolean| = |Int| * |Boolean|
```

A sealed trait is called a Sum type

A sealed trait is called a Sum type

A case class is called a Product type

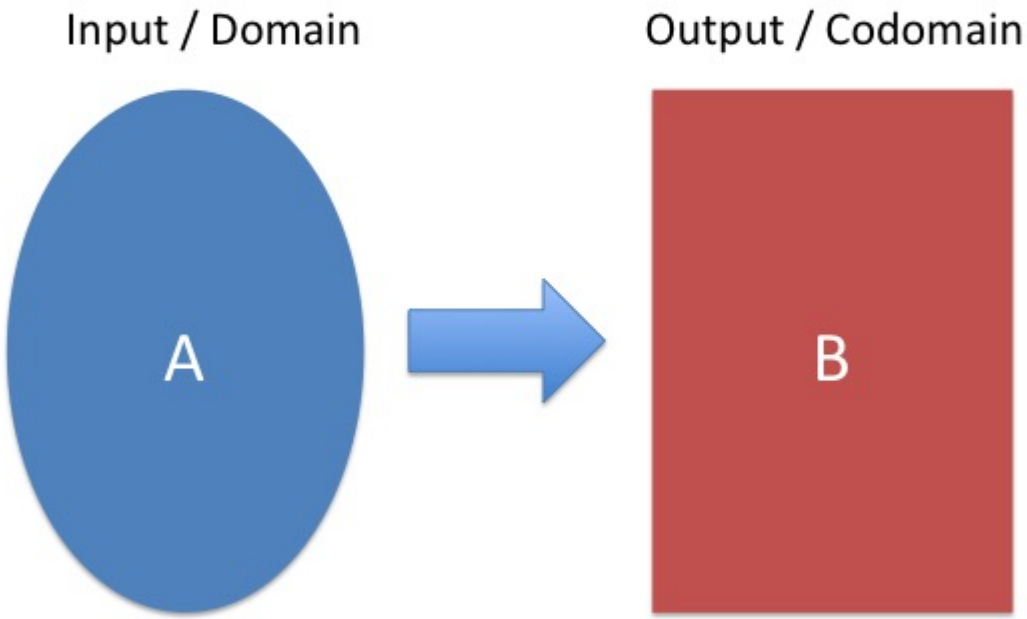
A sealed trait is called a Co-Product

A case class is called a Product

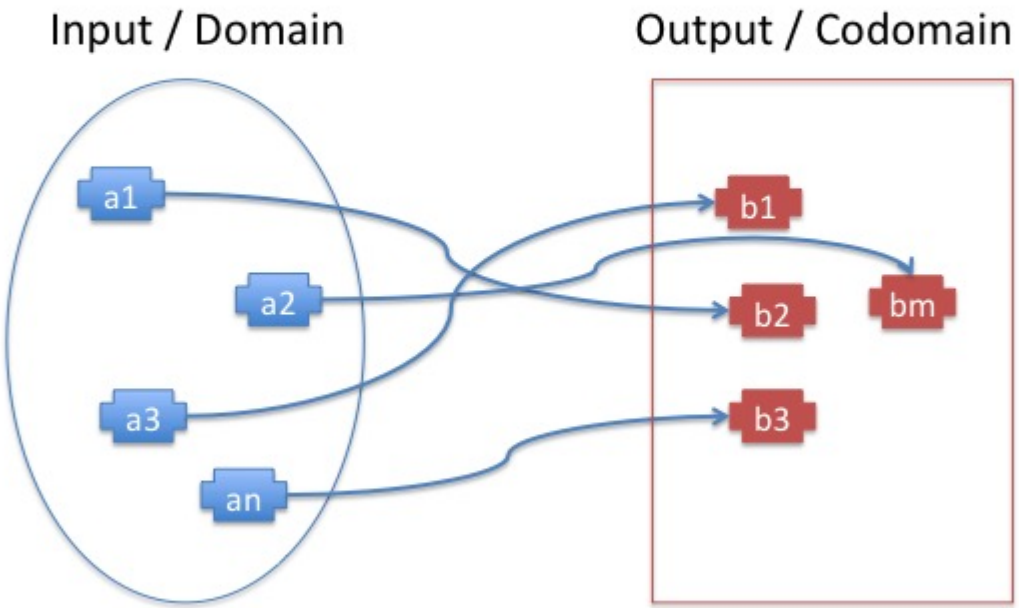
Exercise 2

$$|A => B|$$

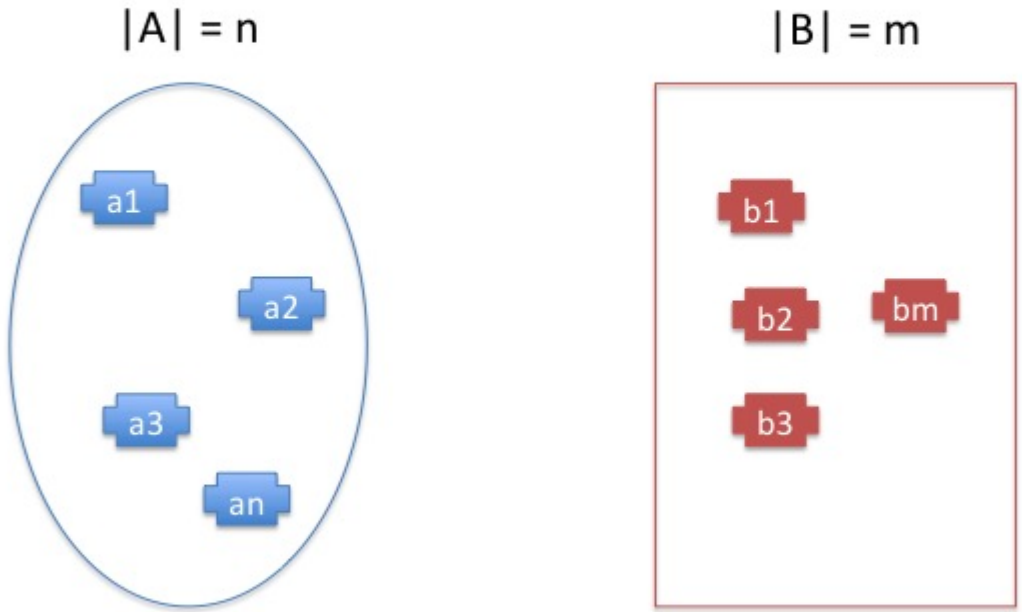
Function is a mapping



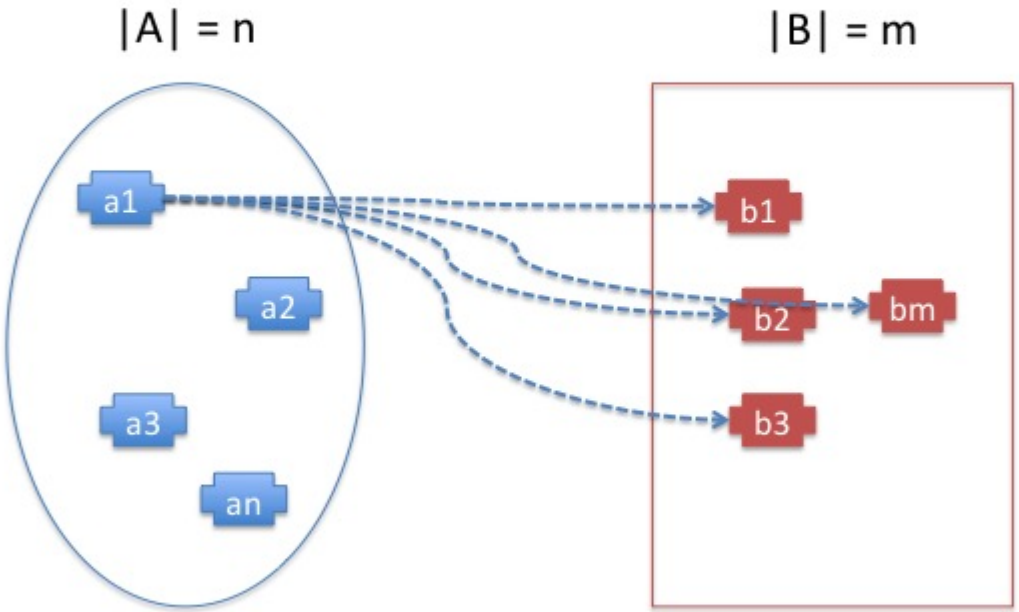
Function is a mapping



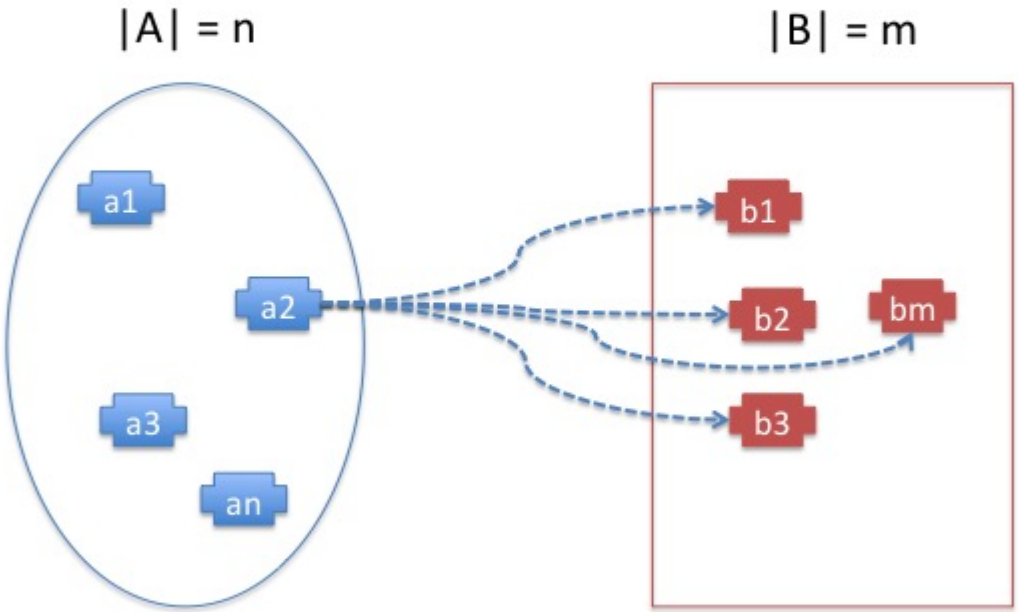
|A => B|



|A => B|

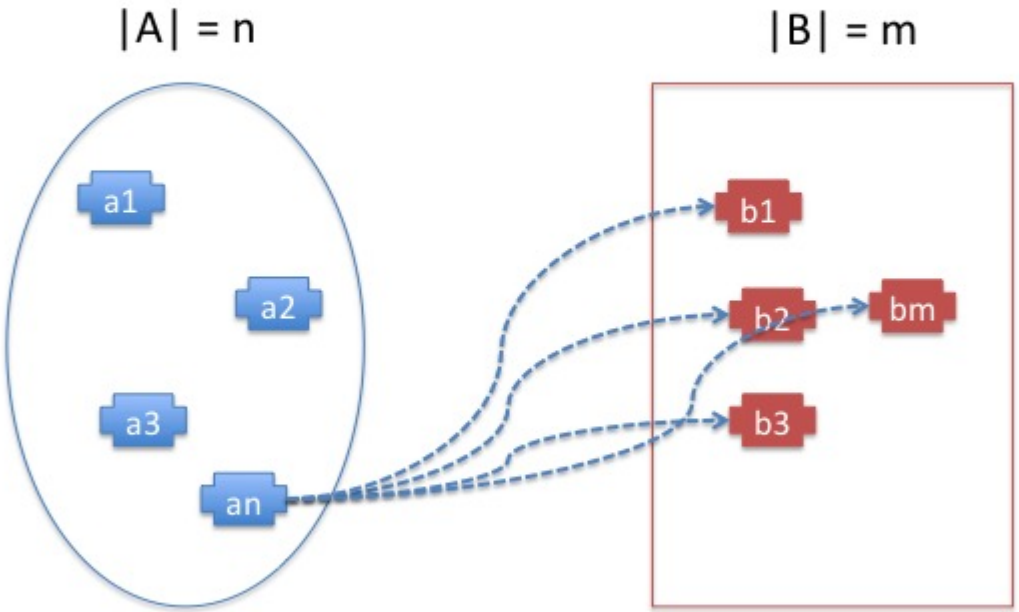


|A => B|



|A => B| = m * m * ...

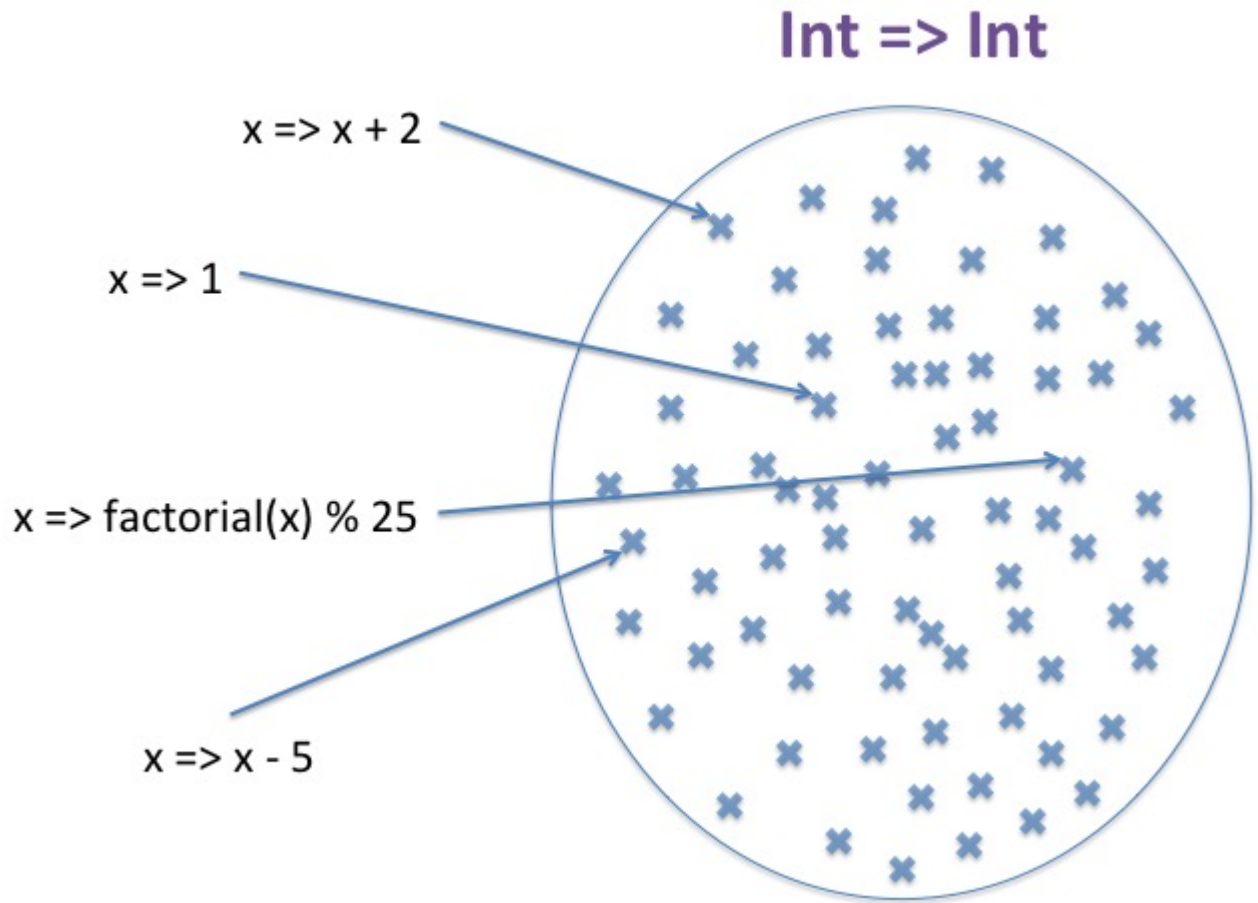
|A => B|



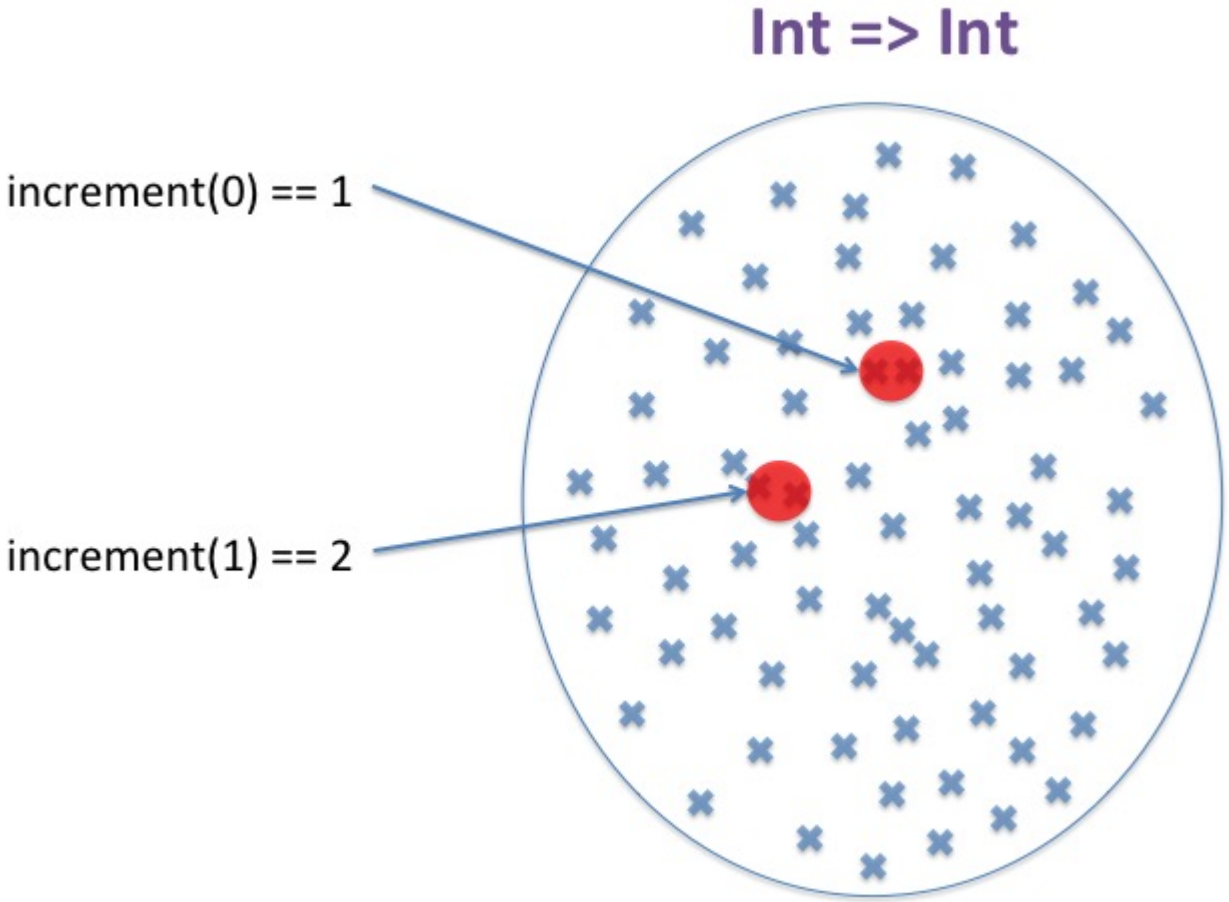
$$|A \Rightarrow B| = m * m * \ldots * m$$
$$= m \wedge n = |B| \wedge |A|$$

Exercise 3

Function is a set!

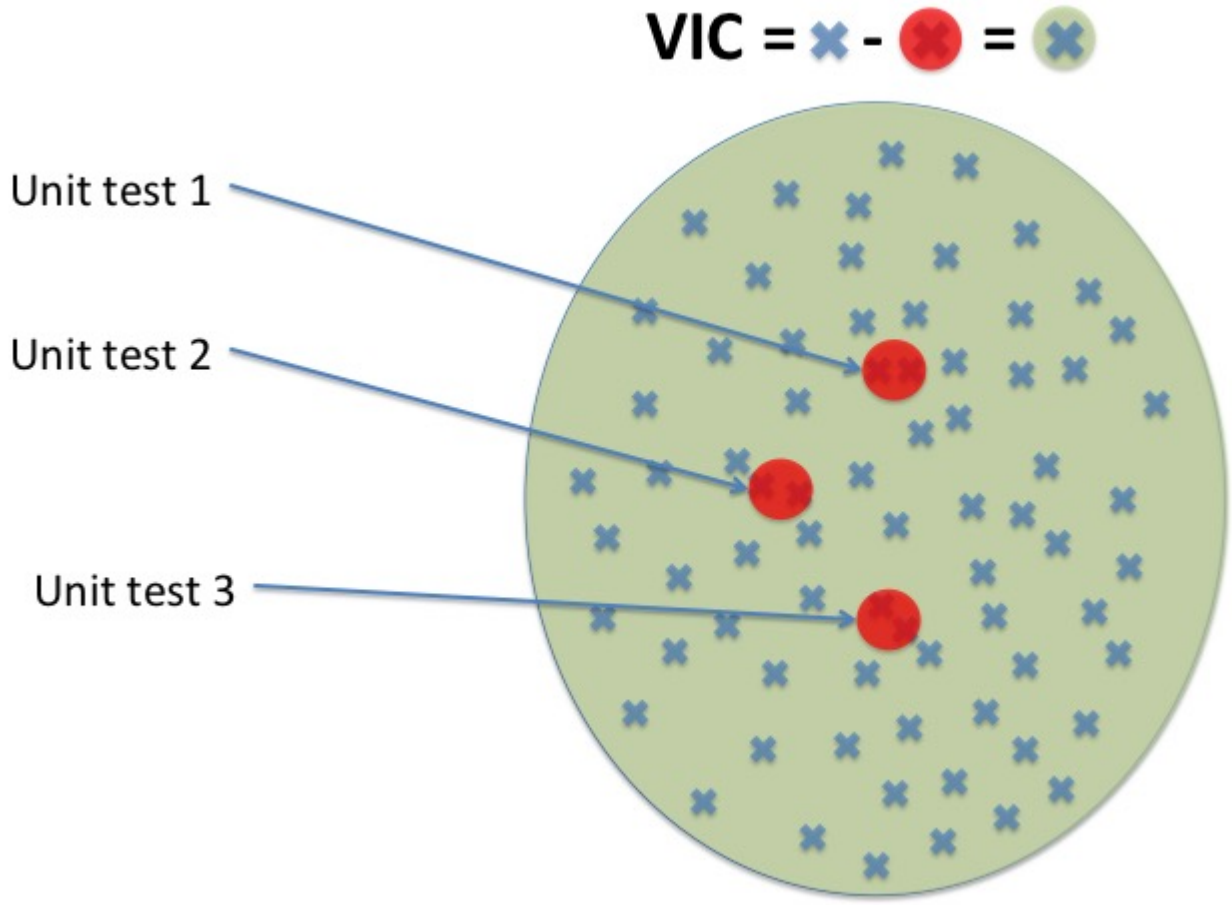


Unit Tests



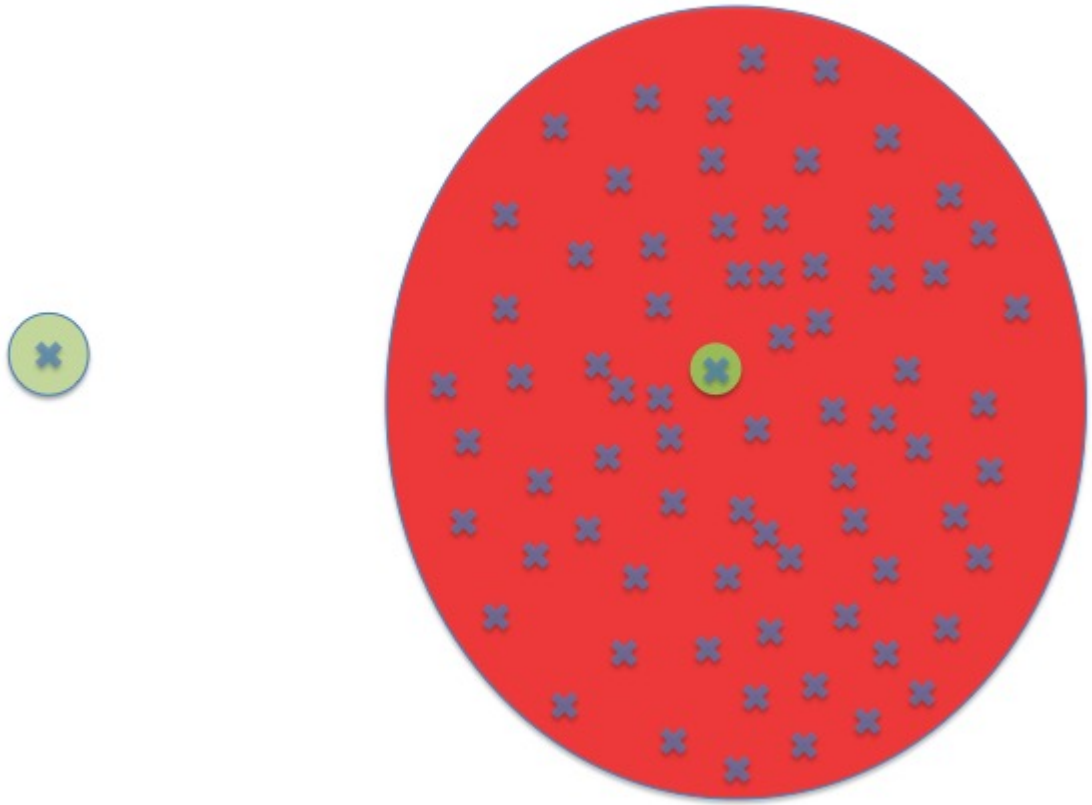
Valid Implementation Count (VIC)

VIC(f: A => B) = number of impl - impl invalid by tests


VIC = - =
Unit test 1
Unit test 2
Unit test 3

The smaller is VIC, the more f is constrained

VIC(f) = 1



VIC(f: A => B) = **number of impl** - impl invalid by tests

$$|A \Rightarrow B| = |B| \hat{} |A|$$

VIC(f: A => B) = number of impl - **impl invalid by tests**

Unit Test

```
sealed trait DayOfWeek

case object Monday  extends DayOfWeek
case object Tuesday extends DayOfWeek
// ...
case object Sunday  extends DayOfWeek
```

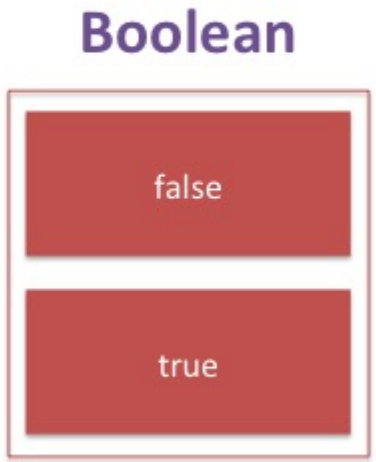
```
val f: DayOfWeek => Boolean = ???
```

such as

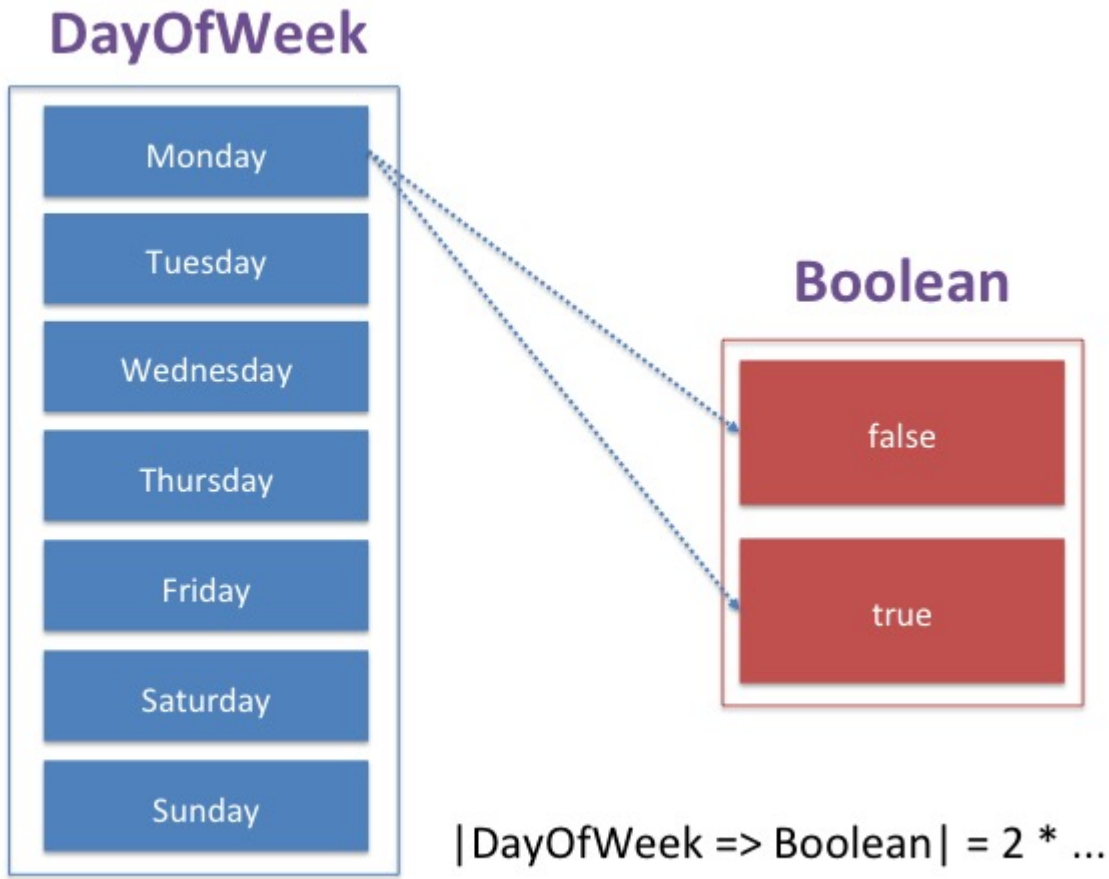
```
assert(f(Tuesday) == false)
```

Unit Test

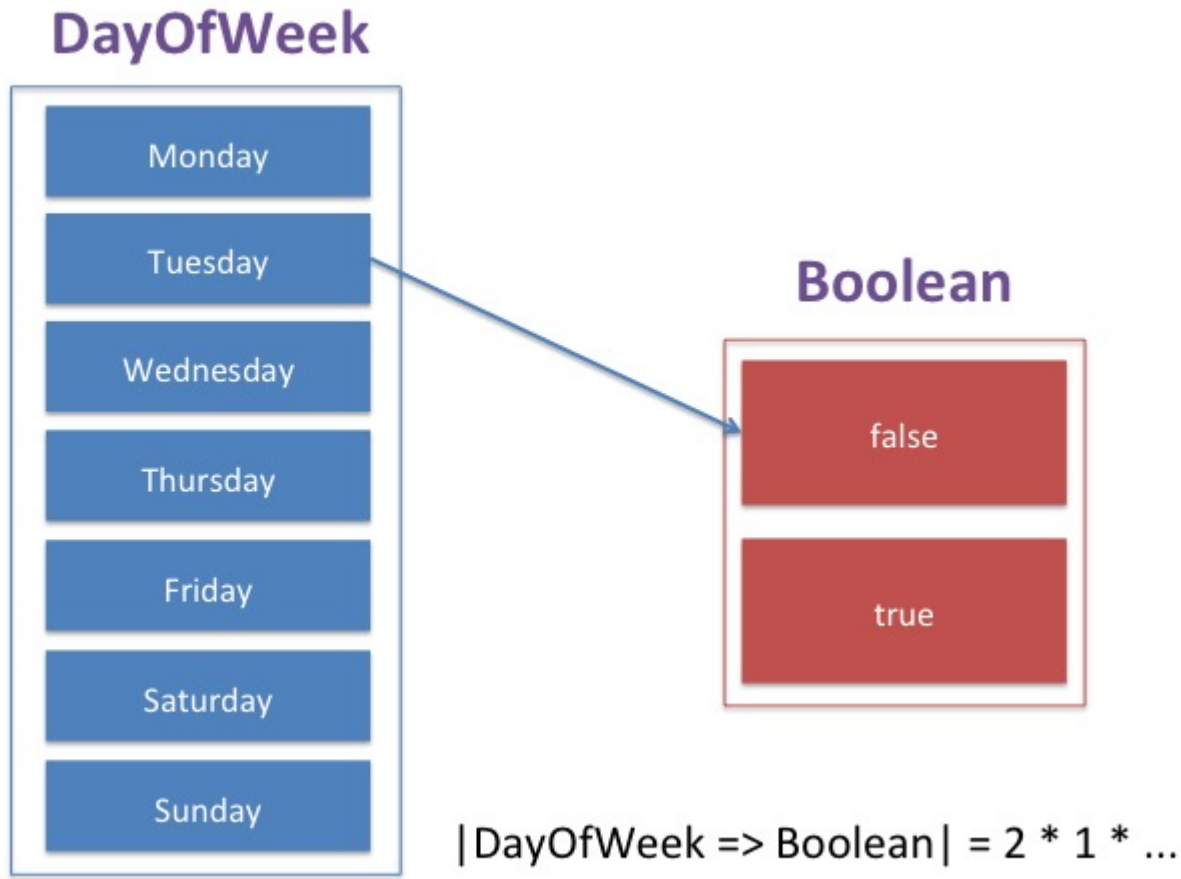

DayOfWeek
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday


Boolean
false
true

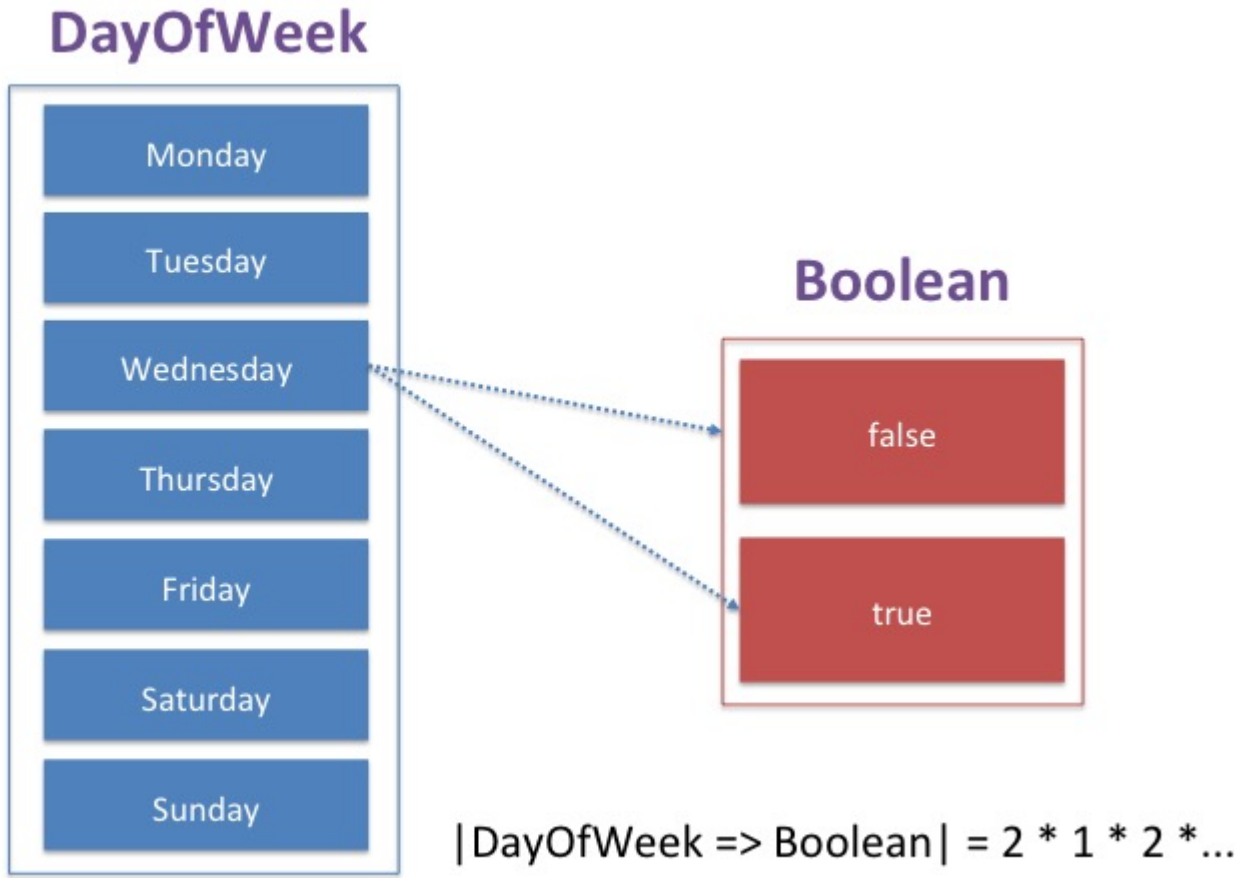
Unit Test


DayOfWeek
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Boolean
false
true
|DayOfWeek => Boolean| = 2 * ...

Unit Test


DayOfWeek
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Boolean
false
true
|DayOfWeek => Boolean| = 2 * 1 * ...

Unit Test


DayOfWeek
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Boolean
false
true
|DayOfWeek => Boolean| = 2 * 1 * 2 *...

One Unit Test

```
VIC(f: A => B) = |A => B| / |B|
               = |B| ^ |A| / |B|
               = |B| ^ (|A| - 1)
```

One Unit Test

```
VIC(f: A => B) = |A => B| / |B|
               = |B| ^ |A| / |B|
               = |B| ^ (|A| - 1)
```

Two Unit Tests

```
VIC(f: A => B) = |A => B| / |B| / |B|
               = |B| ^ |A| / |B| / |B|
               = (|B| ^ |A|) / (|B| ^ 2)
               = |B| ^ (|A| - 2)
```

$$VIC(f: A => B) = |B| \hat{} (|A| - n)$$

where n is the number of unit tests

Exercise 4 and 5

Type Algebra

Type	Algebra
Nothing	0
Unit	1
Either[A, B]	A + B
(A, B)	A * B
A => B	B ^ A
Isomorphism	A == B

Curry–Howard isomorphism

Propositions as types from Philip Wadler

Type Algebra Logic

Type	Algebra	Logic
Nothing	0	⊥ (false)
Unit	1	⊤ (true)
Either[A, B]	A + B	A ∨ B (or)
(A, B)	A * B	A ∧ B (and)
A => B	B ^ A	A → B (implies)
Isomorphism	A == B	A ⇔ B (equivalence)

Type Algebra Logic

Type	Algebra	Logic
Nothing	0	⊥
Unit	1	⊤
Either[A, B]	A + B	A ∨ B
(A, B)	A * B	A ∧ B
A => B	B ^ A	A → B
Isomorphism	A == B	A ⇔ B

```
Either[A, Nothing] == A
```

```
Either[A, Nothing] == A
```

```
A ∨ ⊥ ⇔ A
```

```
(A, Nothing) == Nothing
```

```
(A, Nothing) == Nothing
```

```
A ∧ ⊥ ⇔ ⊥
```

Find the representation that makes sense to you

```
Either[Int, String] => Boolean     <===>     (Int => Boolean, String => Boolean)
```

Find the representation that makes sense to you

```
Either[A, B] => C   <===>   (A => C, B => C)
```

Find the representation that makes sense to you

```
Either[A, B] => C   <===>   (A => C, B => C)
```

Algebra

```
Either[A, B] => C = C ^ (A + B)
                  = C ^ A * C ^ B
                  = (A => C, B => C)
```

Logic

```
Either[A, B] => C = (A ∨ B) → C
                  = (A → C) ∧ (B → C)
                  = (A => C, B => C)
```

Summary

- Cardinality of types matter
- Unit tests offer almost no benefit in term of correctness
- `VIC(f: A => B) = |B| ^ (|A| - n)`
- Two techniques to achieve correctness
  - Property based testing
  - Parametric polymorphism

Resources and further study

- Programming with Algebra: property based testing with storage
- Choosing properties for property-based testing
- Property-Based Testing in a Screencast Editor
- Property-Based Testing The Ugly Parts: Case Studies from Komposition
- Types vs Tests
- Counting type inhabitants
- Thinking with types: type, algebra, logic
- Propositions as types: Curry–Howard isomorphism

Module 3: Typeclass