


FOUNDATION



Type

Cost of misusing types

```
case class Country(value: String)

val UK: Country      = Country("United Kingdom")
val France: Country  = Country("France")
val Germany: Country = Country("Germany")
```



Cost of misusing types

```
case class Country(value: String)

val UK: Country      = Country("United Kingdom")
val France: Country  = Country("France")
val Germany: Country = Country("Germany")
```

```
def getCurrency(country: Country): String = ???
```

such as

```
getCurrency(Country("United Kingdom")) == "GBP"
getCurrency(Country("France"))          == "EUR"
getCurrency(Country("Germany"))         == "EUR"
```



Cost of misusing types

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



Cost of misusing types

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

```
scala> getCurrency(Country("UK"))  
res0: Option[String] = None
```

```
scala> getCurrency(Country("GBR"))  
res1: Option[String] = None
```

```
scala> getCurrency(Country("Royaume-Uni"))  
res2: Option[String] = None
```



Cost of misusing types

```
sealed trait Country

object Country {
  case object UnitedKingdom extends Country
  case object France         extends Country
  case object Germany        extends Country
}
```

```
import Country._

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

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



Cost of misusing types

```
sealed trait Country
object Country {
  case object UnitedKingdom extends Country
  case object France        extends Country
  case object Germany       extends Country
}

sealed trait Currency
object Currency {
  case object BritishPound extends Currency
  case object Euro         extends Currency
}
```

```
import Country._, Currency._

def getCurrency(country: Country): Currency =
  country match {
    case UnitedKingdom => BritishPound
    case France | Germany => Euro
  }
```



Plan

- What is the cost of misusing types
- How to use ADTs to encode data
- Learn how to measure impact of types and tests
- Explore relationship between types, algebra and logic



Exercise 1: Misused types

`exercises.types.TypeExercises.scala`



Type should **exactly** fit business requirements



Imprecise data lead to errors
and misleading documentation



How should we encode data?



Algebraic Data Type (ADT)

- OR, a ConfigValue is
 - a String OR
 - an Int OR
 - Empty
- AND, a User is
 - an userId AND
 - a name AND
 - an address



OR

- a Boolean is true OR false
- an Int is a -10 OR 0 OR 1 OR ...
- a DayOfTheWeek is Monday OR Tuesday OR Wednesday OR ...
- an Option is a Some OR a None
- a List is a Nil OR a Cons
- a Json is a JsonNumber OR a JsonString OR a JsonArray OR a JsonObject OR ...



How should we encode OR?

A ConfigValue is a String OR an Int OR Empty



OR Encoding

```
sealed trait ConfigValue

object ConfigValue {
  case class ConfigString(value: String) extends ConfigValue
  case class ConfigNumber(value: Double) extends ConfigValue
  case object ConfigEmpty extends ConfigValue
}
```



OR Encoding

```
sealed trait ConfigValue

object ConfigValue {
  case class ConfigString(value: String) extends ConfigValue
  case class ConfigNumber(value: Double) extends ConfigValue
  case object ConfigEmpty extends ConfigValue
}
```

In Scala 3

```
enum ConfigValue {
  case ConfigString(value: String)
  case ConfigNumber(value: Double)
  case ConfigEmpty
}
```



AND

- a User is a userId AND a name AND an address
- a ZonedDateTime is a dateTime AND a timeZone
- a Cons is a head AND a tail
- a Tuple2 is a _1 AND a _2



How should we encode AND?

A User is a `userId` AND a `name` AND an `address`



AND Encoding

```
import java.util.UUID

case class User(userId: UUID, name: String, address: Address)

case class Address(streetNumber: Int, streetName: String, postCode: String)
```

```
scala> User(UUID.randomUUID(), "John Doe", Address(108, "Cannon Street", "EC4N 6EU"))
res3: User = User(d23dfb6c-0a8b-4b2f-aa3f-3199fc4b0f96,John Doe,Address(108,Cannon Street,EC4N 6EU))
```



Exercise 2: Data Encoding

`exercises.types.TypeExercises.scala`



Case class and sealed trait map exactly to business language AND and OR

Together, they form what is called Algebraic Data Types (ADTs)



Nested AND and OR can be used to encode data precisely

OR is generally underused



How can we compare two encodings?

```
def getCurrency(country: String): Option[String]
```

Is it better to reduce input or reduce output?

```
def getCurrency(country: Country): String
```

```
def getCurrency(country: String): Option[Currency]
```

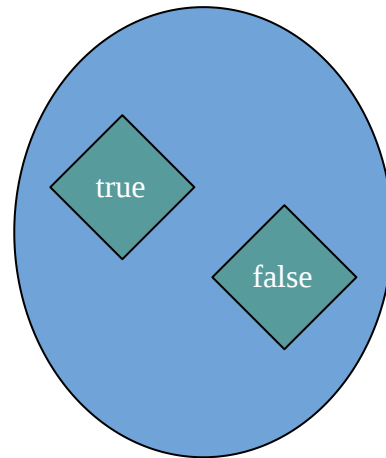
How much better it is to reduce both?

```
def getCurrency(country: Country): Currency
```

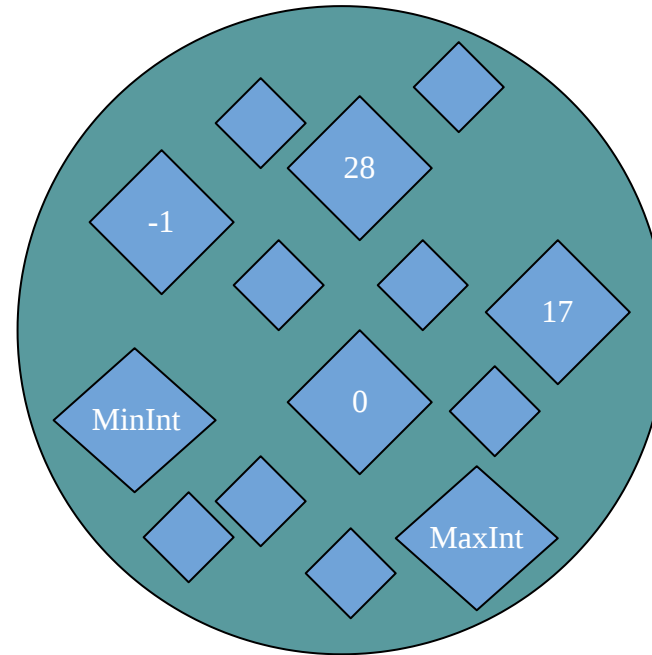


Type is a set

Boolean

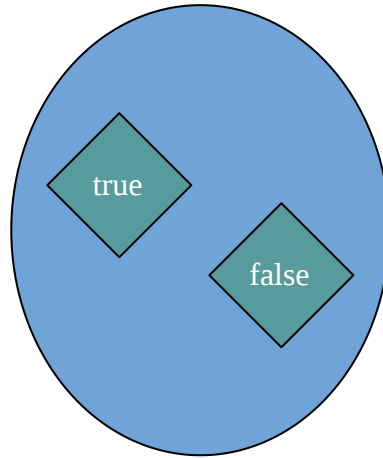


Int

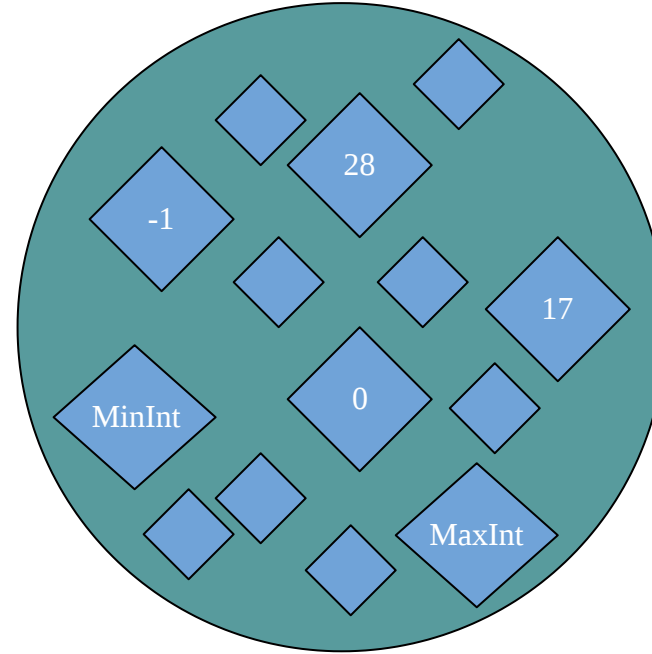


Cardinality

$$|\text{Boolean}| = 2$$



$$|\text{Int}| = 2^{32}$$

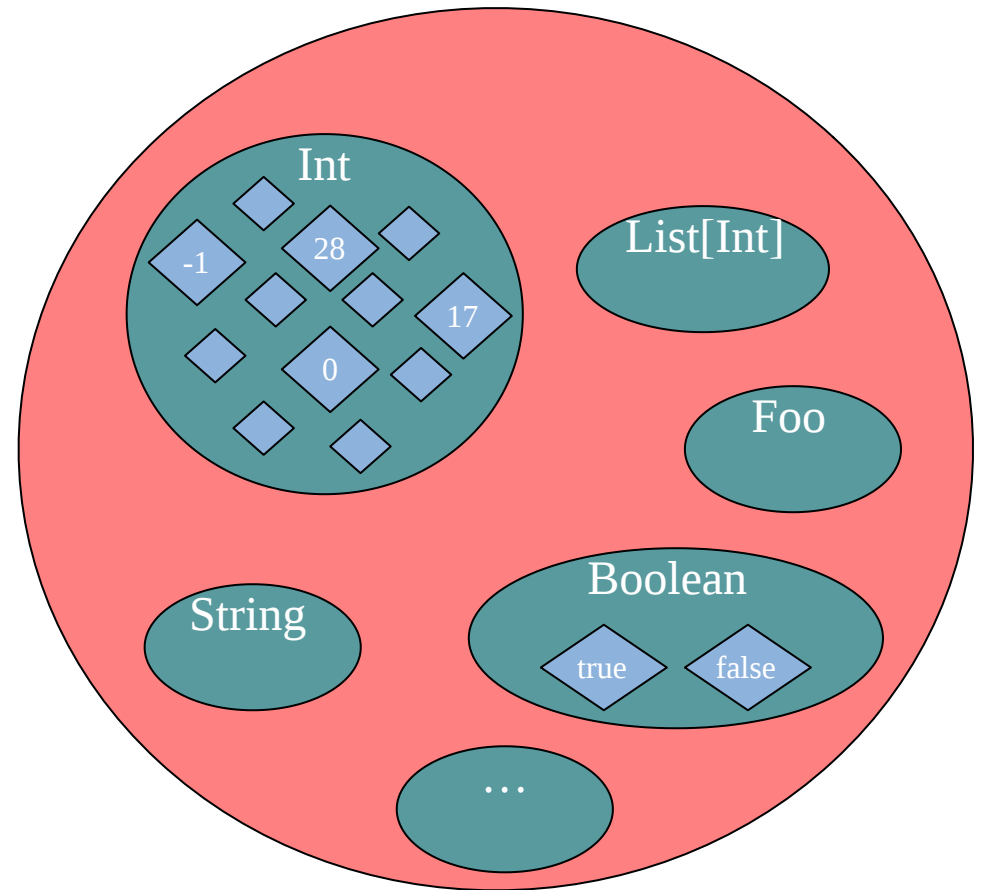


Cardinality: Any

```
val x: Int = 3  
val hello: String = "hello"  
case class User(name: String, age: Int)  
val john: User = User("John", 53)
```

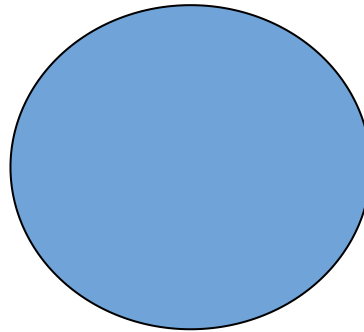
```
scala> x: Any  
res4: Any = 3  
  
scala> john: Any  
res5: Any = User(John,53)  
  
scala> List(x, hello, john)  
res6: List[Any] = List(3, hello, User(John,53))
```

$$|\text{Any}| = \infty$$



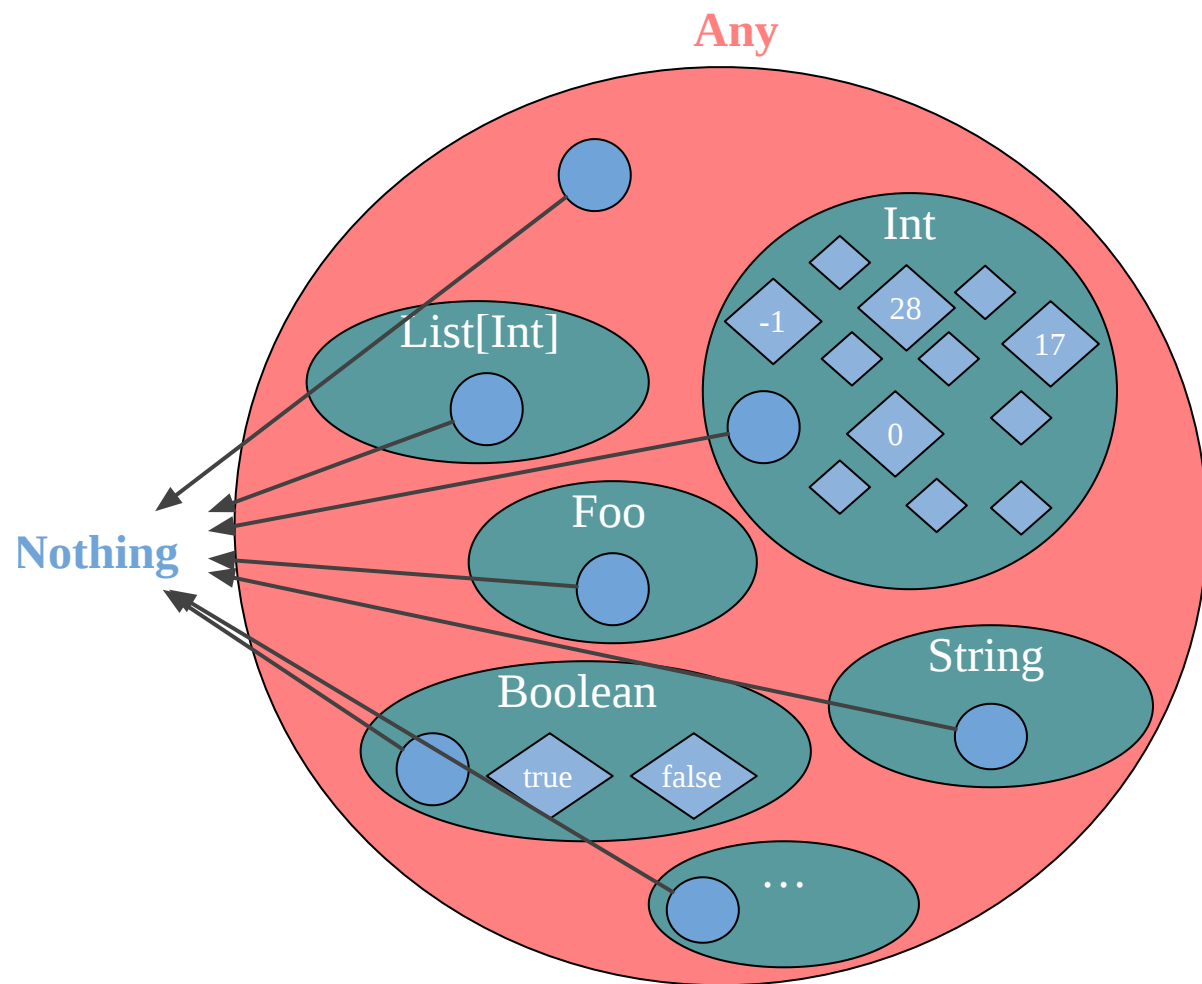
Cardinality: Nothing

$$|\text{Nothing}| = 0$$



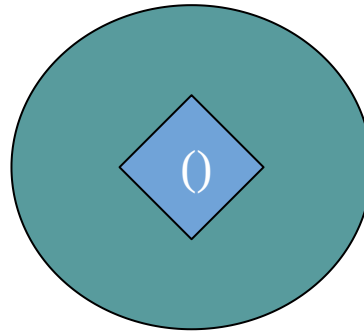
Cardinality: Nothing

```
sealed trait Nothing  
  extends Int  
  with Boolean  
  with String  
  with Foo  
  with List[Int]  
  with Any
```

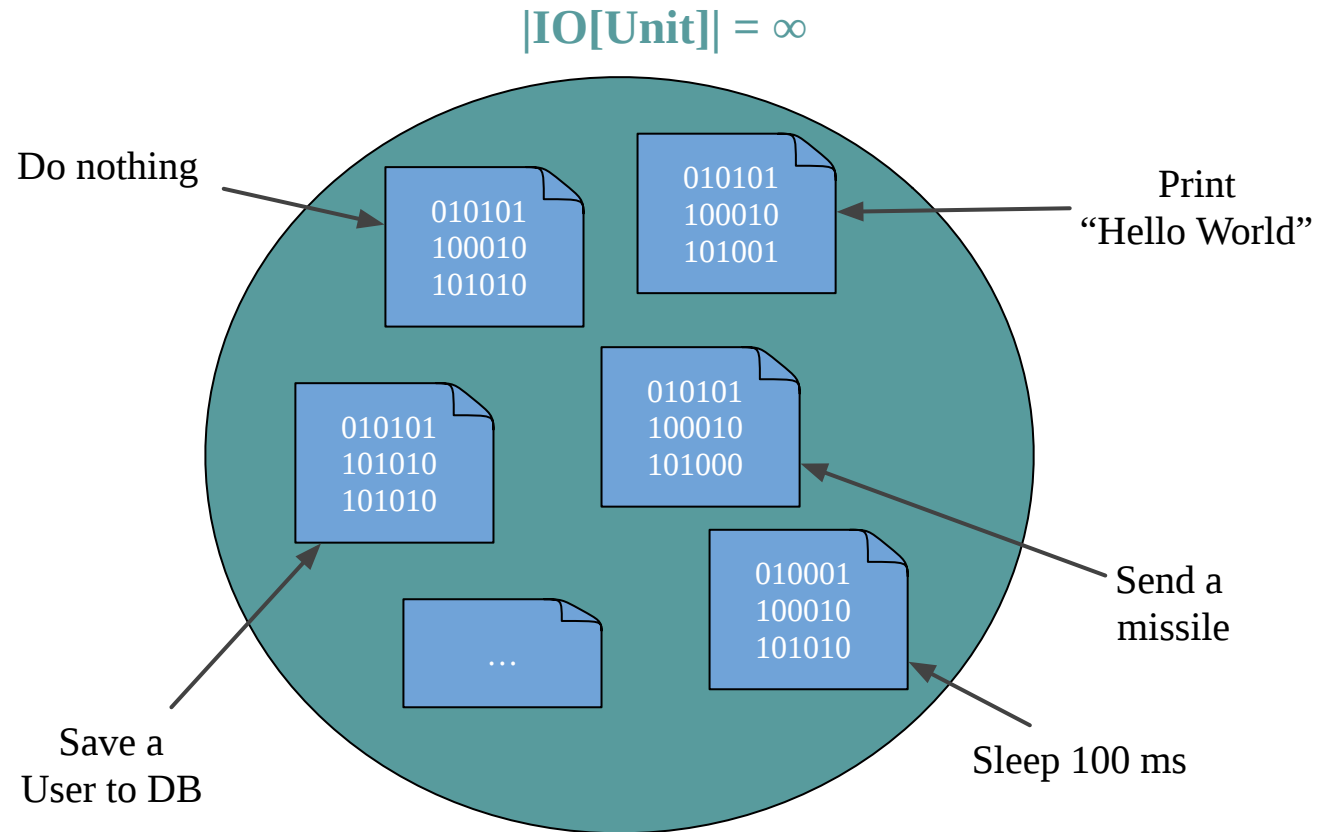


Cardinality: Unit

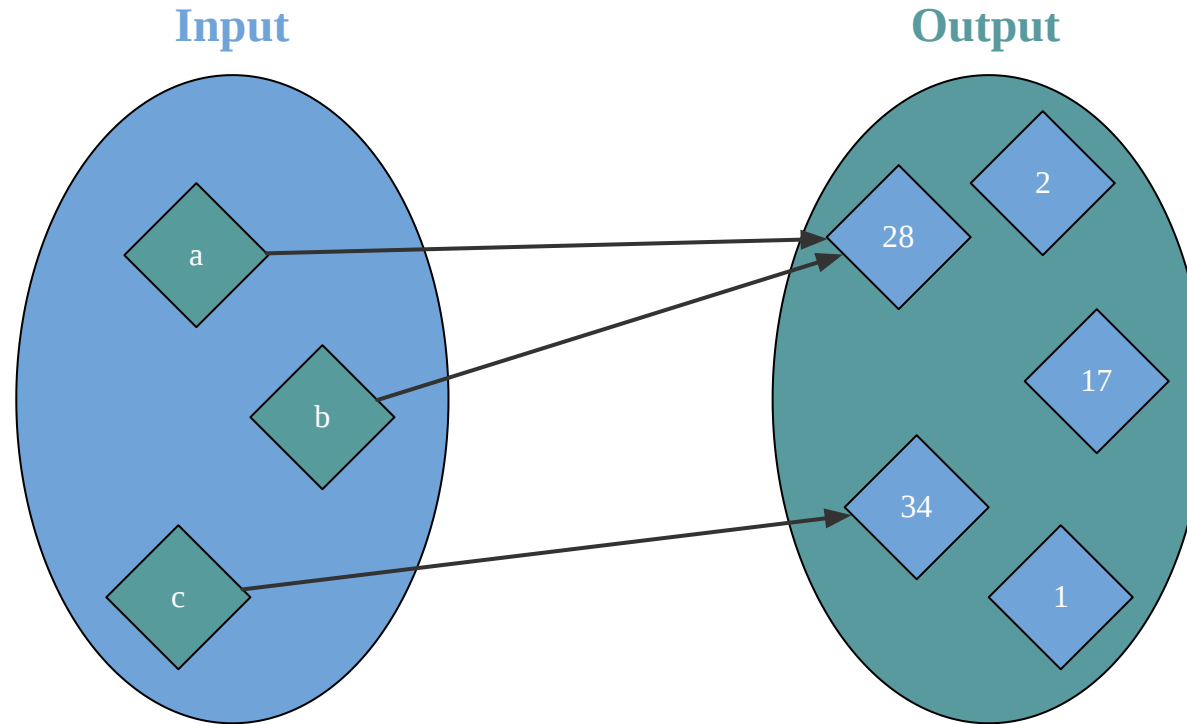
$$|\mathbf{Unit}| = 1$$



Cardinality: IO



Function is mappings between two sets



A => B is a type

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

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



$A \Rightarrow B$ is a type

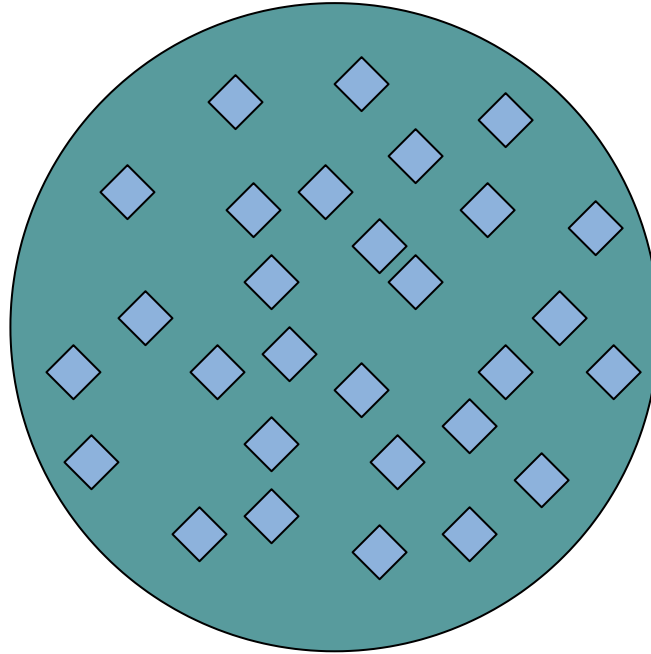
$\&\&$

A type is a set of values

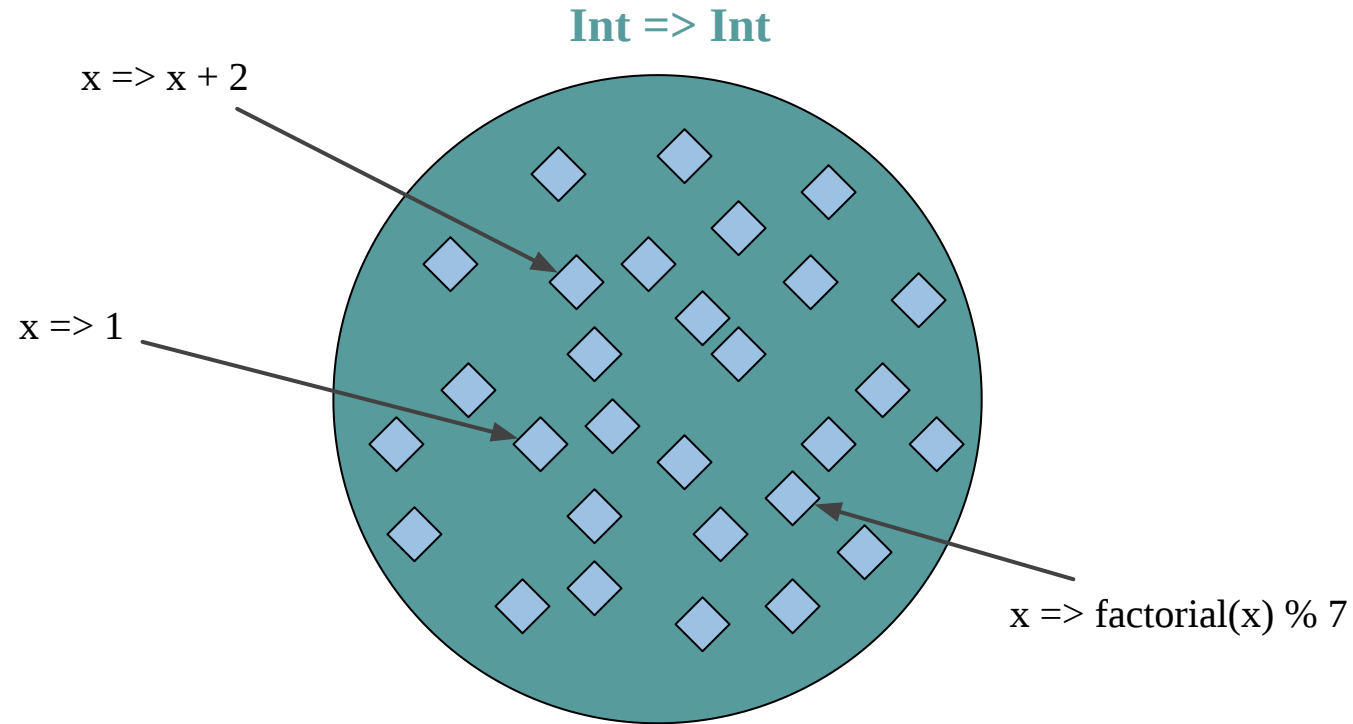


$A \Rightarrow B$ is a set of values!

$A \Rightarrow B$



A function type is a set of implementations!



How many elements are in the set $A \Rightarrow B$?

How many implementations satisfy the type checker?



$$|A \Rightarrow B|$$



The smaller $|A \Rightarrow B|$, the better



Perfect case is when $|A \Rightarrow B| = 1$



Exercise 3: Cardinality

`exercises.types.TypeExercises.scala`



Sealed trait

```
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

```
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 case class is called a **product** type



$$|A \text{ OR } B \text{ OR } C| = |A| + |B| + |C|$$

$$|A \text{ AND } B \text{ AND } C| = |A| * |B| * |C|$$



Exercise 4a-f: Advanced Cardinality

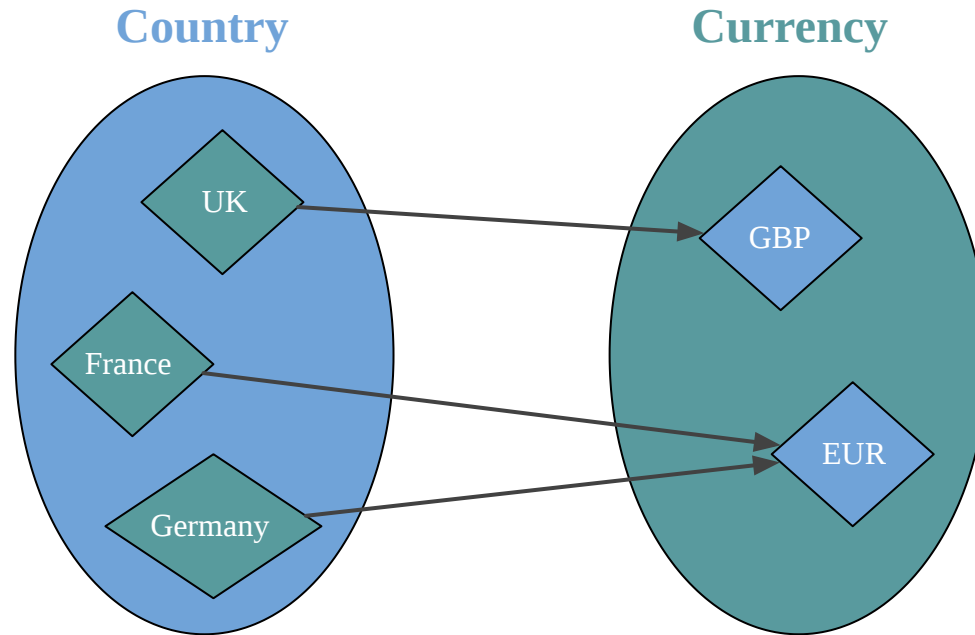
`exercises.types.TypeExercises.scala`



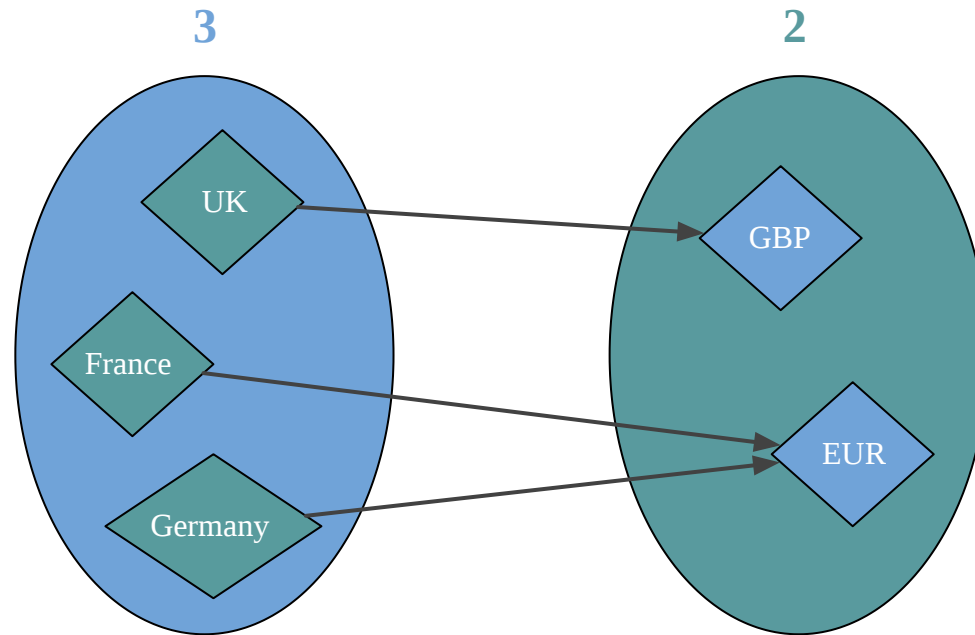
$$|A \Rightarrow B|$$



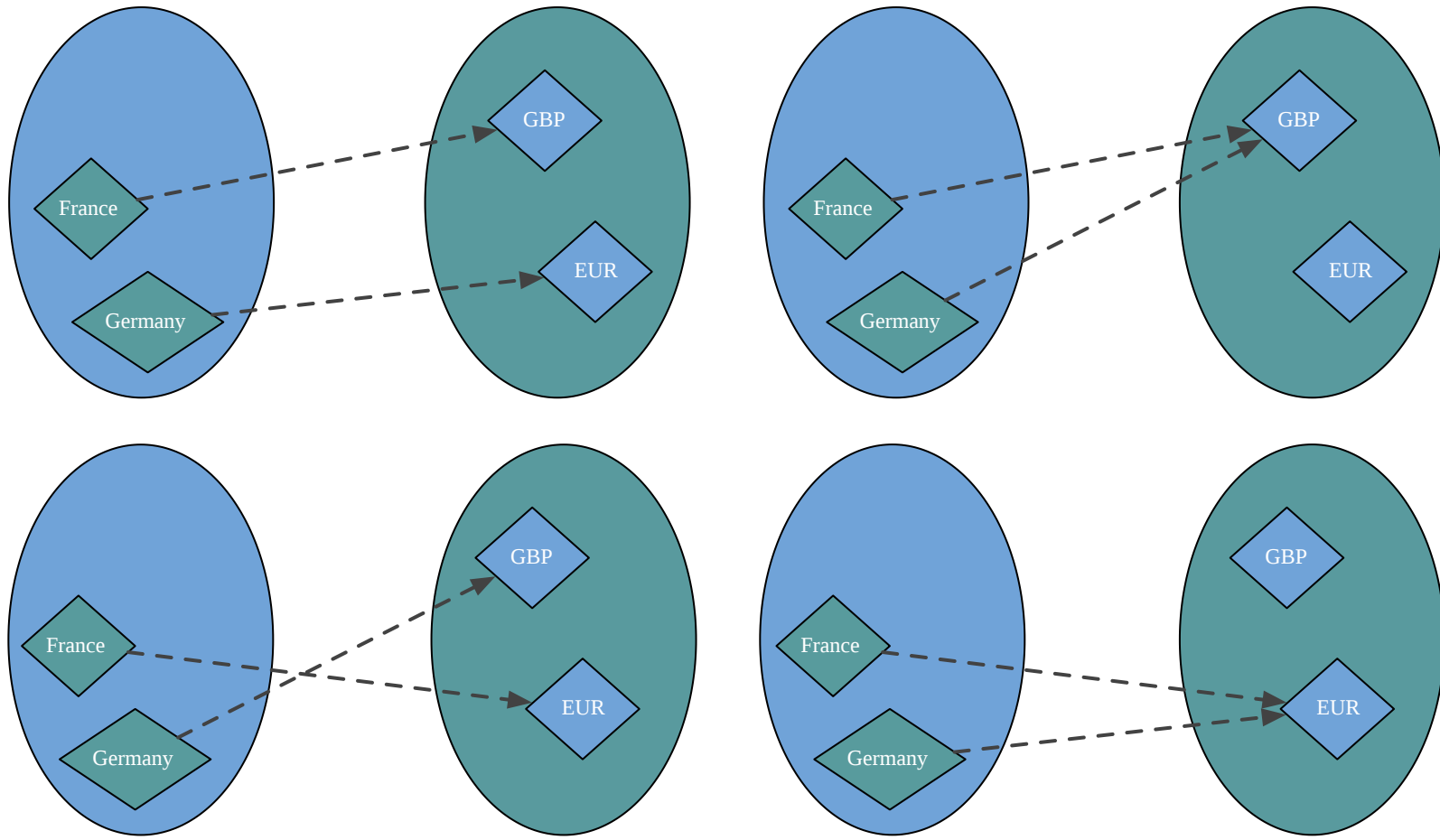
|Country => Currency|



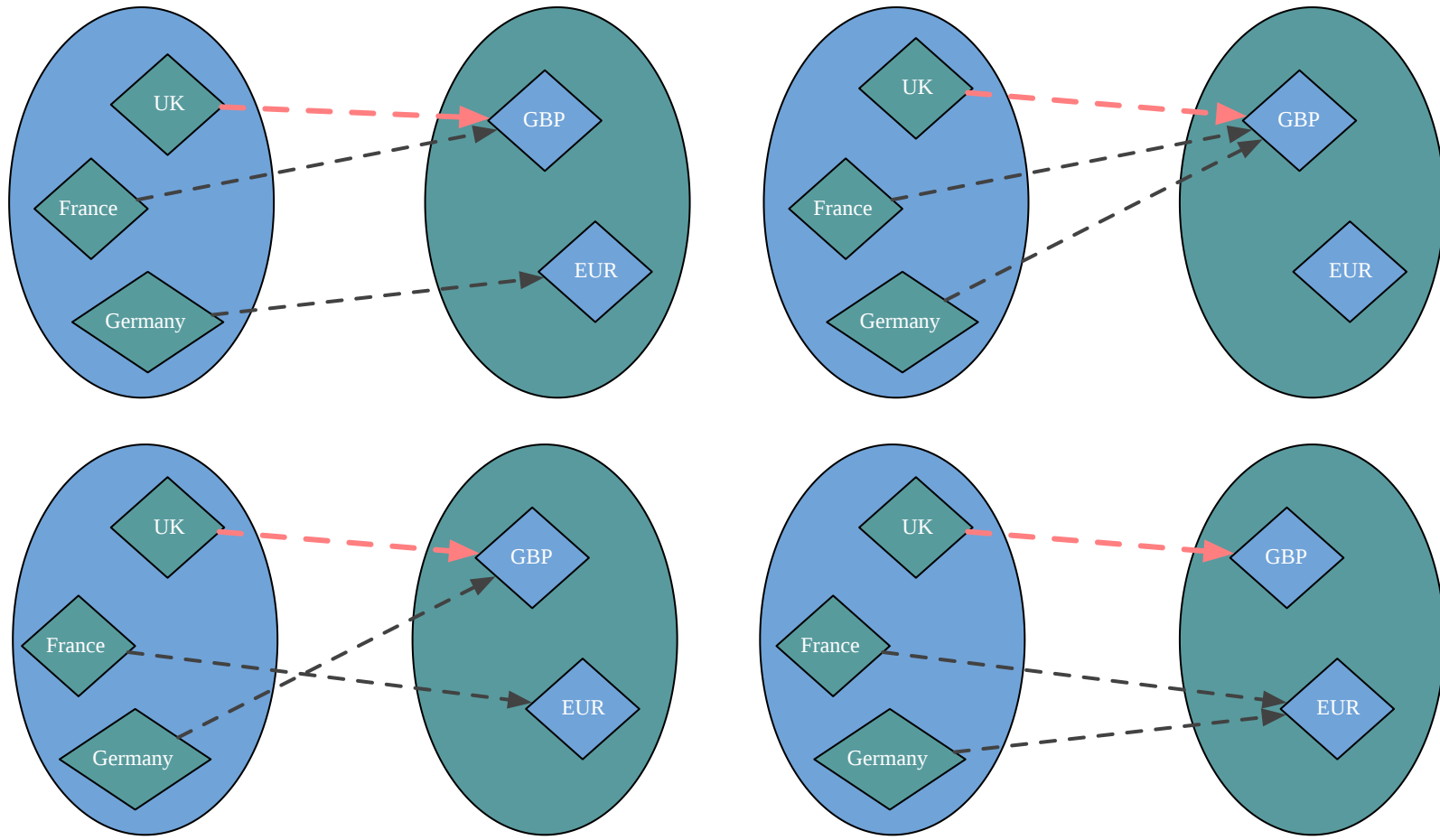
| 3 => 2 |



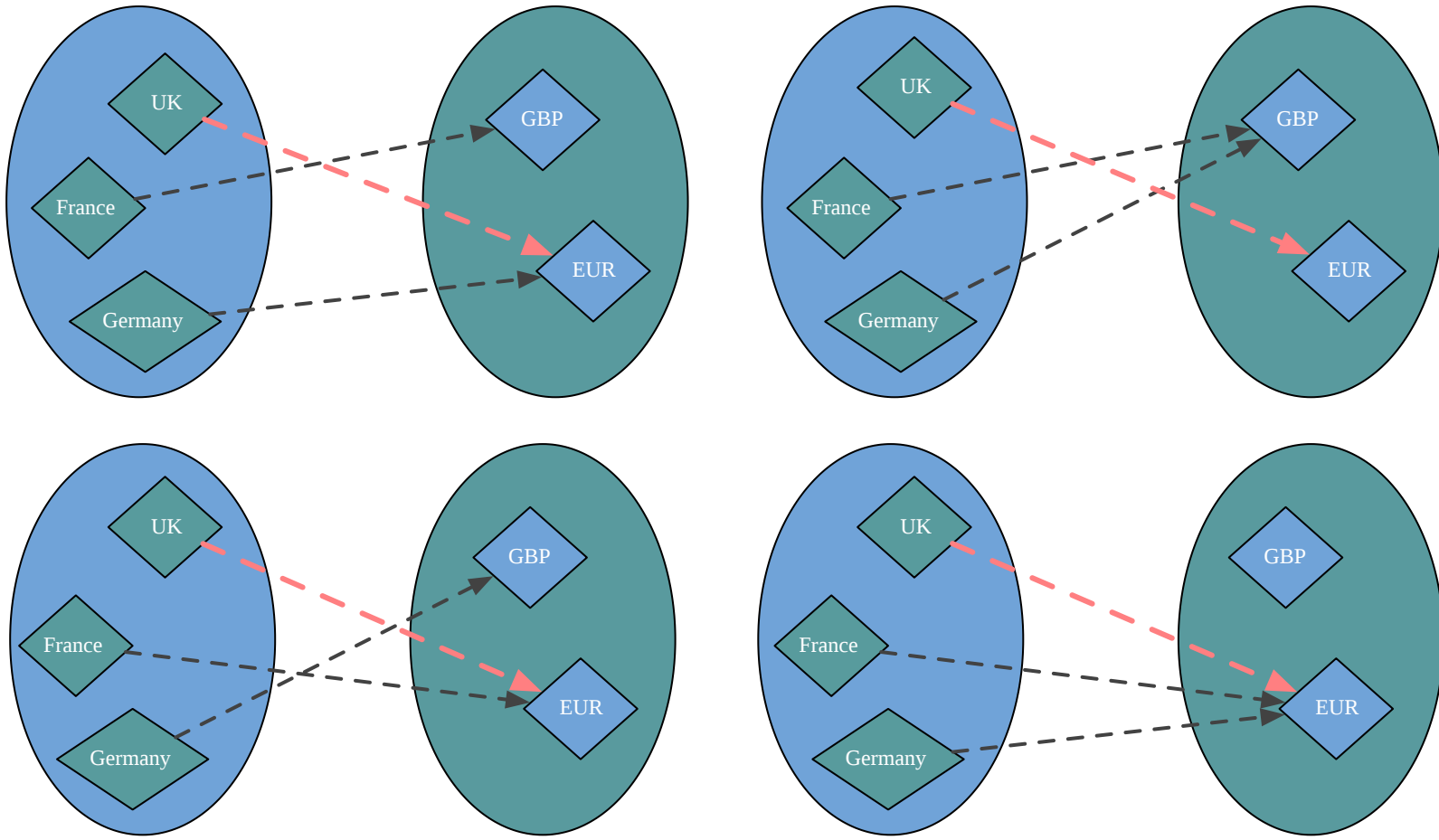
$$|2 \Rightarrow 2| = 4$$



| 3 => 2 |



$$|3 \Rightarrow 2| = |2 \Rightarrow 2| + |2 \Rightarrow 2|$$



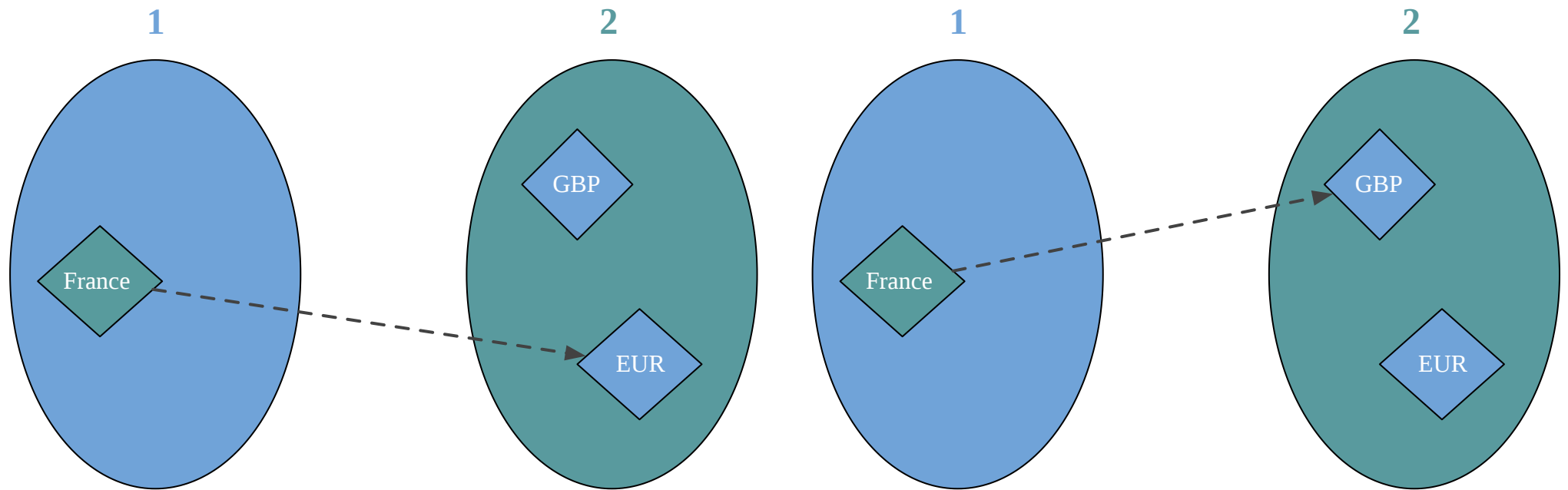
$$|3 \Rightarrow 2| = 2 * |2 \Rightarrow 2|$$



$$\begin{aligned} |3 \Rightarrow 2| &= 2 * |2 \Rightarrow 2| \\ &= 2 * 2 * |1 \Rightarrow 2| \end{aligned}$$



$$|1 \Rightarrow 2| = 2$$



$$\begin{aligned} |3 \Rightarrow 2| &= 2 * |2 \Rightarrow 2| \\ &= 2 * 2 * |1 \Rightarrow 2| \\ &= 2 * 2 * 2 \end{aligned}$$



$$|3 \Rightarrow 2| = 2 * |2 \Rightarrow 2|$$

$$= 2 * 2 * |1 \Rightarrow 2|$$

$$= 2 * 2 * 2$$

$$= 2 ^ 3$$



$$|A \Rightarrow B| = |B| \wedge |A|$$

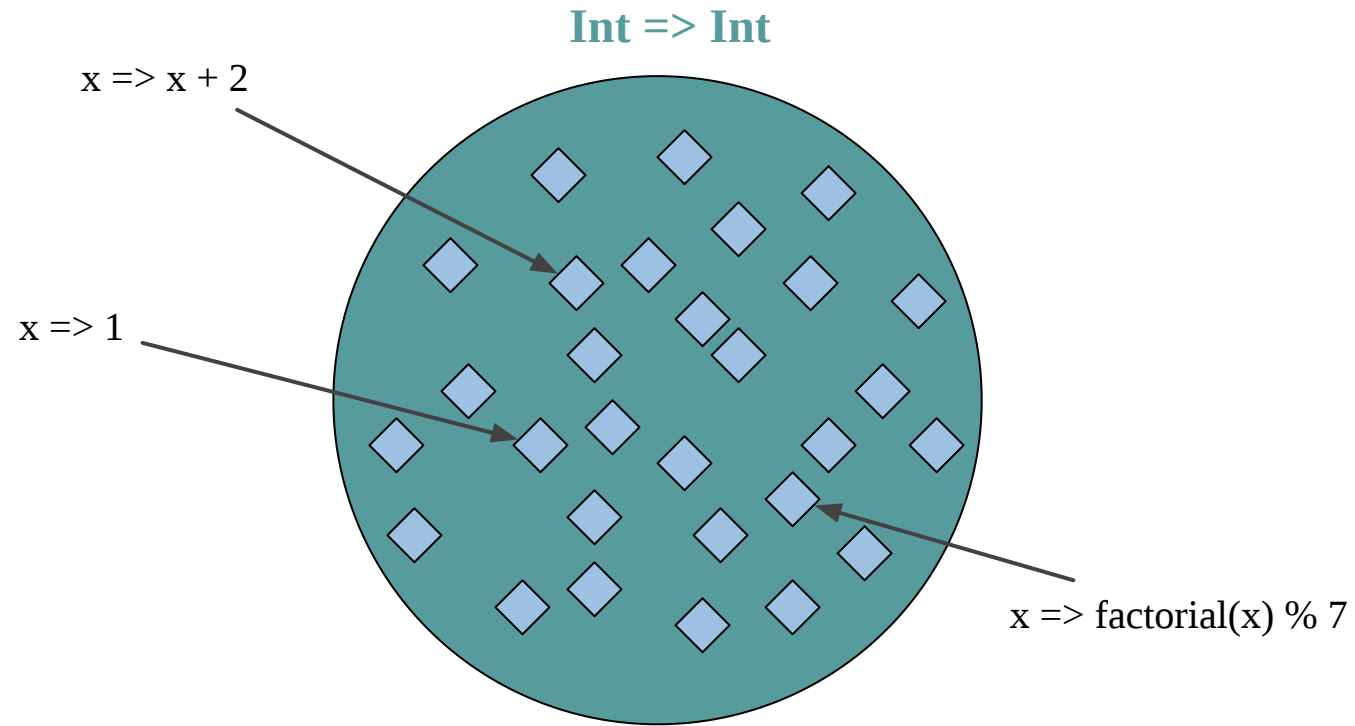


Finish Exercise 4

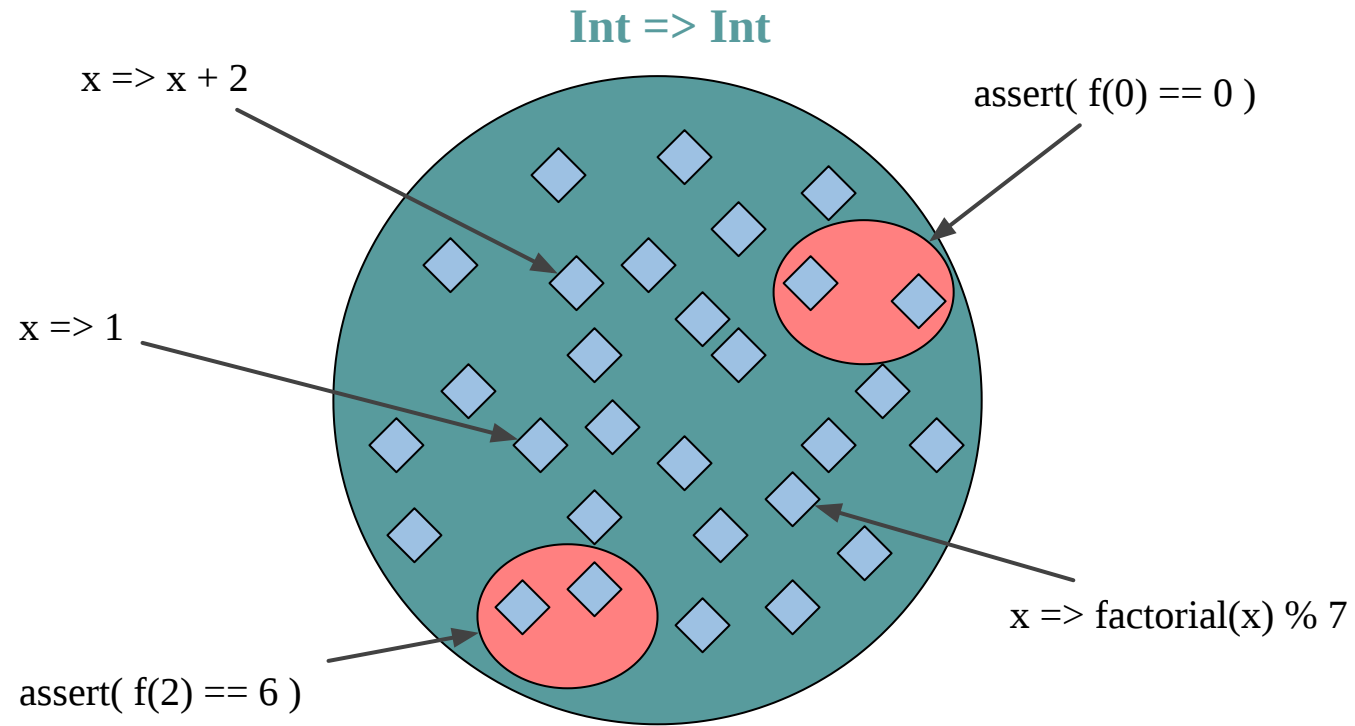
`exercises.types.TypeExercises.scala`



Functions are sets!



Unit tests

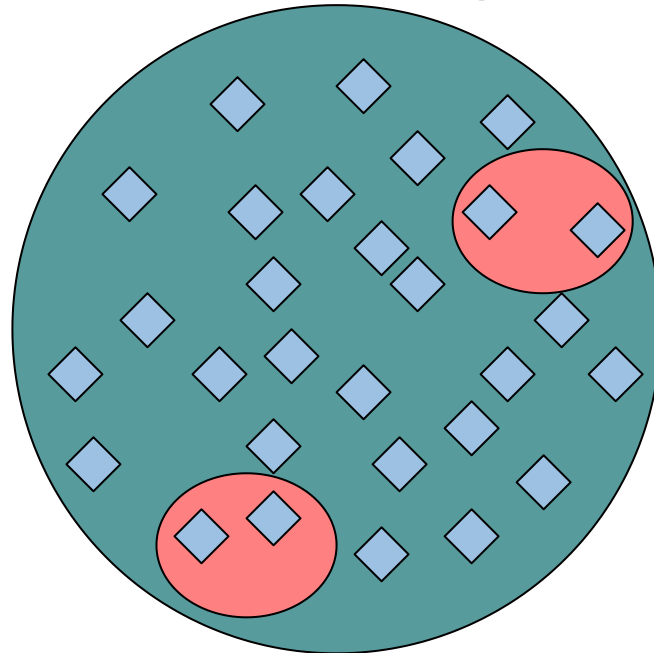


Valid Implementation Count (VIC)

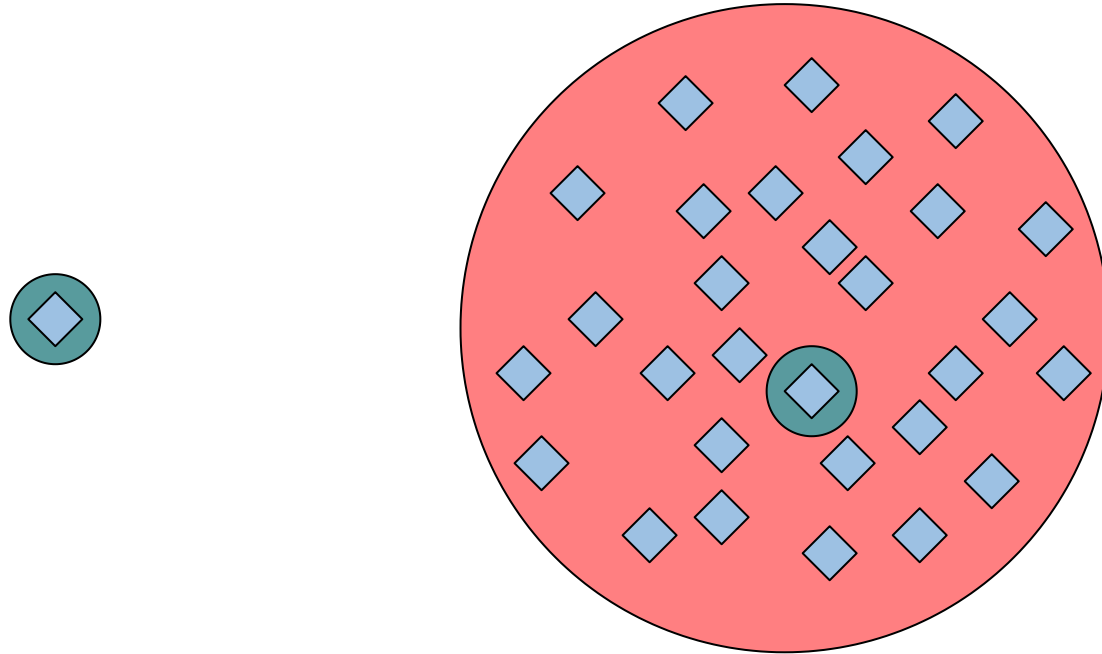




$$\text{VIC} = \begin{matrix} \swarrow & \searrow \\ |A \Rightarrow B| & ??? \end{matrix}$$



$$\text{VIC}(f) = 1$$



Exercise 5: Tests

`exercises.types.TypeExercises.scala`



Unit Test

```
sealed trait Country
object Country {
  case object UnitedKingdom extends Country
  case object France         extends Country
  case object Germany        extends Country
}

sealed trait Currency
object Currency {
  case object GBP extends Currency
  case object EUR extends Currency
}

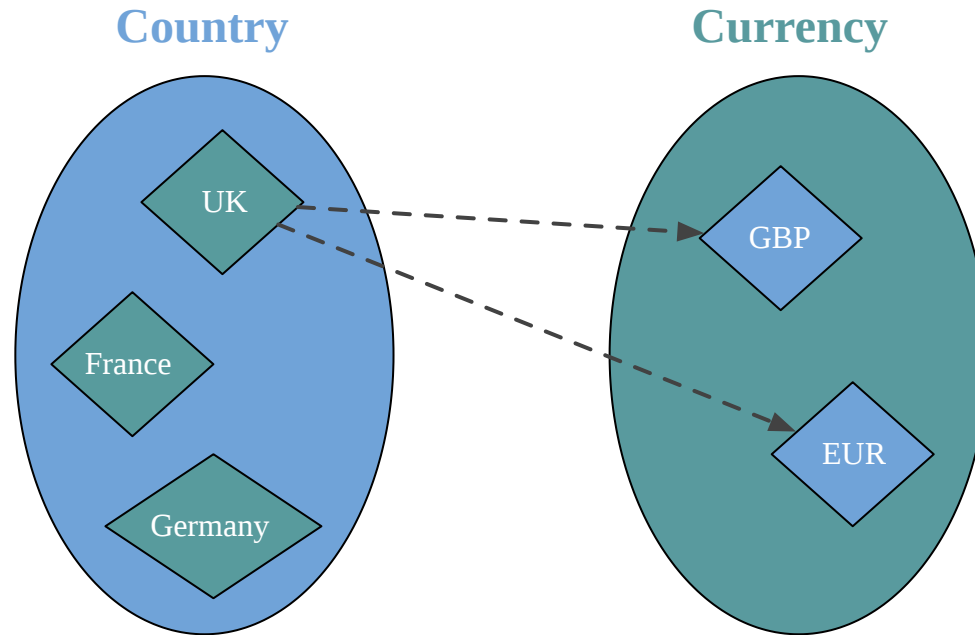
def getCurrency(country: Country): Currency = ???
```

such as

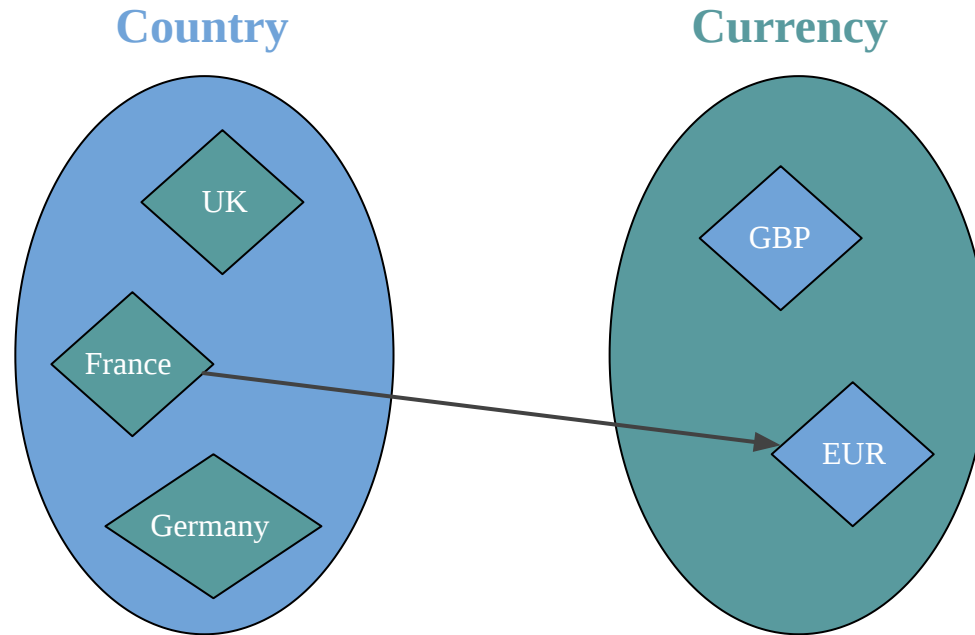
```
assert(getCurrency(France) == EUR)
```



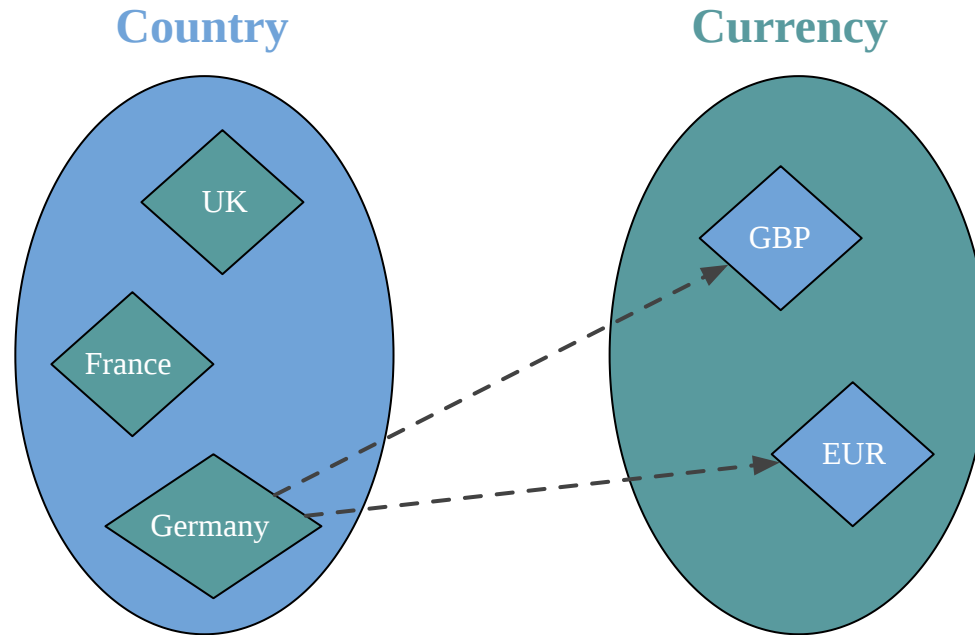
$VIC(\text{getCurrency}) = 2 * \dots$



$VIC(\text{getCurrency}) = 2 * 1 * \dots$



$$\text{VIC}(\text{getCurrency}) = 2 * 1 * 2$$



$$\text{VIC}(f: A \Rightarrow B) = |B| \wedge (|A| - n)$$

where n is the number of unit tests



Exercises 6 and 7

`exercises.types.TypeExercises.scala`



Type Algebra

Type	Algebra
Nothing	0
Unit	1
Either[A, B]	$A + B$
(A, B)	$A * B$
$A \Rightarrow B$	$B \wedge A$
Isomorphism	$A == B$



Curry–Howard isomorphism

[Propositions as types](#) from Philip Wadler



Type Algebra Logic

Type	Algebra	Logic
Nothing	0	\perp
Unit	1	\top
Either[A, B]	$A + B$	$A \vee B$
(A, B)	$A * B$	$A \wedge B$
$A \Rightarrow B$	$B \wedge A$	$A \rightarrow B$
Isomorphism	$A == B$	$A \Leftrightarrow B$



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



`Either[A, Nothing] == A`

`A ∨ ⊥ ⇔ A`



`(A, Nothing) == Nothing`



$(A, \text{Nothing}) == \text{Nothing}$

$A \wedge \perp \Leftrightarrow \perp$



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

$$\text{Either}[A, B] \Rightarrow C \iff (A \Rightarrow C, B \Rightarrow C)$$

Algebra

$$\begin{aligned}\text{Either}[A, B] \Rightarrow C &= C \wedge (A + B) \\ &= C \wedge A * C \wedge B \\ &= (A \Rightarrow C, B \Rightarrow C)\end{aligned}$$

Logic

$$\begin{aligned}\text{Either}[A, B] \Rightarrow C &= (A \vee B) \rightarrow C \\ &= (A \rightarrow C) \wedge (B \rightarrow C) \\ &= (A \Rightarrow C, B \Rightarrow C)\end{aligned}$$



Summary

- Cardinality of types matters
- Unit tests offer almost no benefit in term of correctness
- $VIC(f: A \Rightarrow B) = |B| \wedge (|A| - n)$
- Two techniques to achieve correctness
 - Property based testing
 - Parametric polymorphism



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)  
res7: Any = 6
```

```
scala> inc(10.3)  
res8: Any = 11.3
```

```
scala> inc('c')  
res9: Any = c1
```

```
scala> inc(java.time.Instant.ofEpochMilli(0))  
scala.MatchError: 1970-01-01T00:00:00Z (of class java.time.Instant)  
  at .inc(<console>:2)  
  ... 42 elided
```



$$VIC(Any \Rightarrow Any) = |Any| \wedge (|Any| - n)$$

where n is the number of unit tests



Resources and further study

- [Programming with Algebra](#): property based testing with storage
- [Much Ado About Testing](#): property based testing best practices and pitfalls
- [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 6: Typeclass

