FOUNDATION

Cost of misusing types

```
case class Country(value: String)

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

Cost of misusing types

```
case class Country(value: String)

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

```
getCurrency(Country("UK"))
// res0: Option[String] = None
getCurrency(Country("GBR"))
// res1: Option[String] = None
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 `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 `NonEmptyList` 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)
```

```
User(UUID.randomUUID(), "John Doe", Address(108, "Cannon Street", "EC4N 6EU"))
// res5: User = User(
//   158ab0dc-146e-4c4c-bd9f-0b7280184989,
//   "John Doe",
//   Address(108, "Cannon Street", "EC4N 6EU")
// )
```

Algebraic data types mix AND and OR

```
sealed trait Role

object Role {
  case class Reader(accountId: Long, premiumUser: Boolean) extends Role
  case class Editor(accountId: Long, favoriteFont: Option[String]) extends Role
  case object Admin extends Role
}

case class User(userId: Long, role: Role, address: Option[Address])
```

Exercise 2: Data Encoding

`exercises.types.TypeExercises.scala`

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

Nested AND and OR form what is called Algebraic Data Types (ADTs)

ADT allow to encode data precisely

How can we compare two encodings?

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

Is it better to reduce input or reduce output?

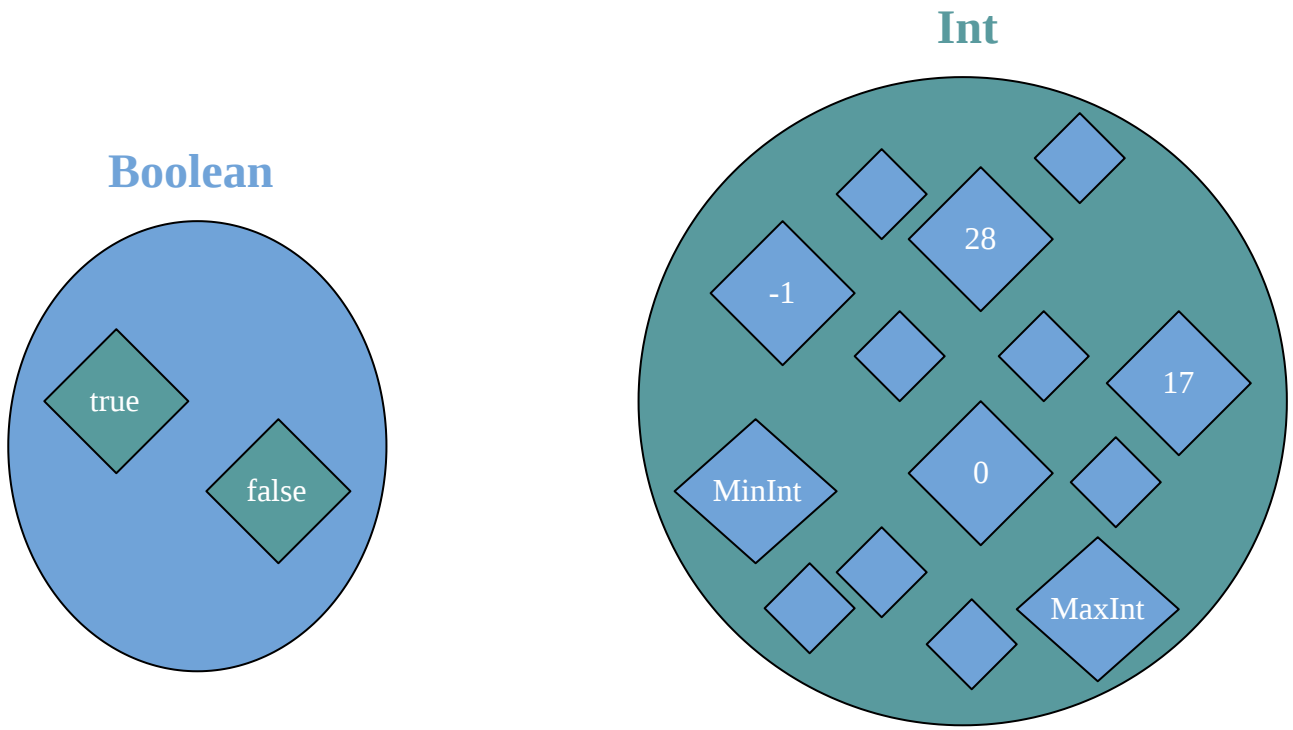
```
def getCurrency(country: Country): String
```

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

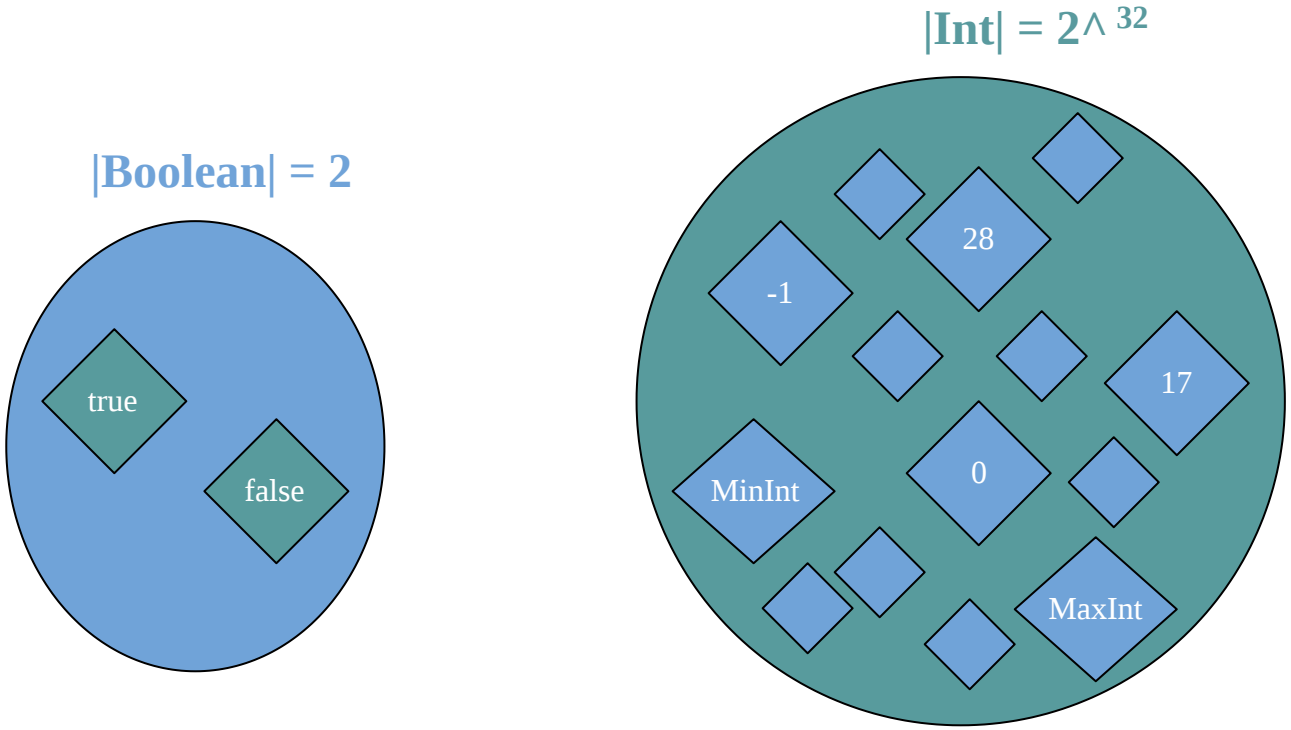
How much better it is to reduce both?

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

Type is a set



Cardinality



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

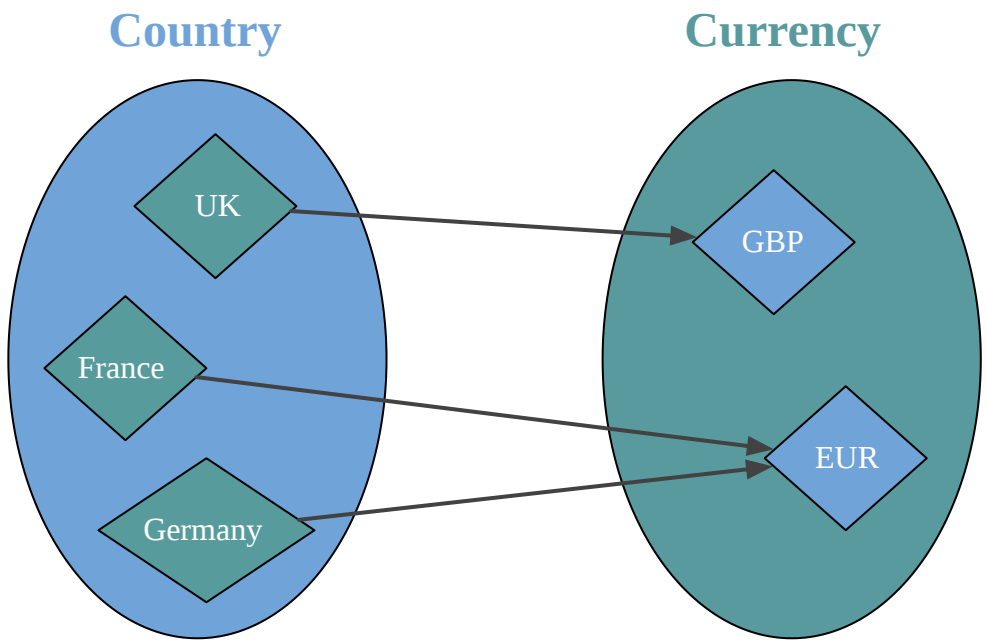
$$|A \text{ OR } B \text{ OR } C| = |A| + |B| + |C|$$

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

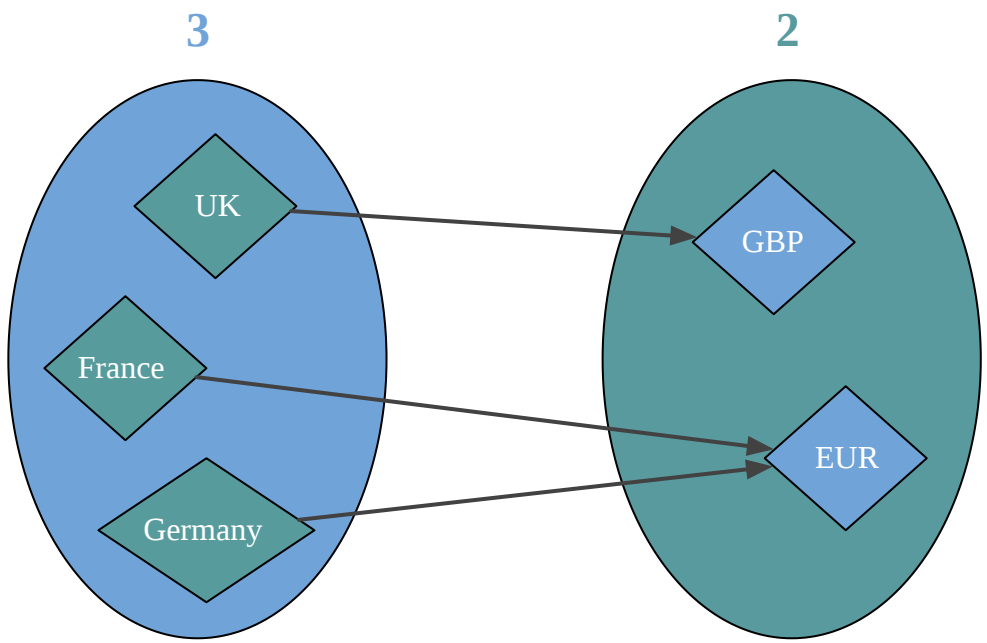
Finish Exercise 3

`exercises.types.TypeExercises.scala`

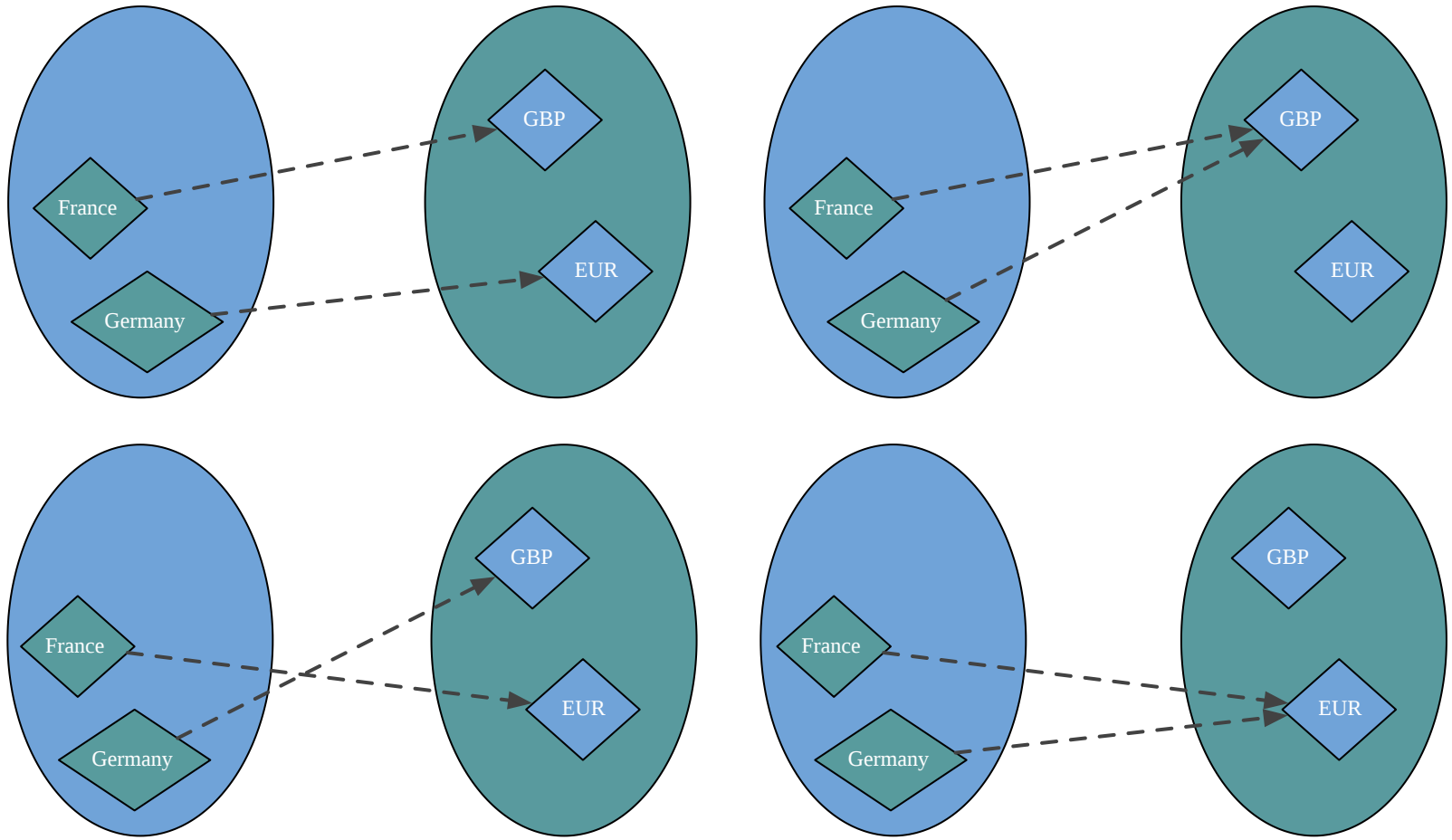
|Country => Currency|



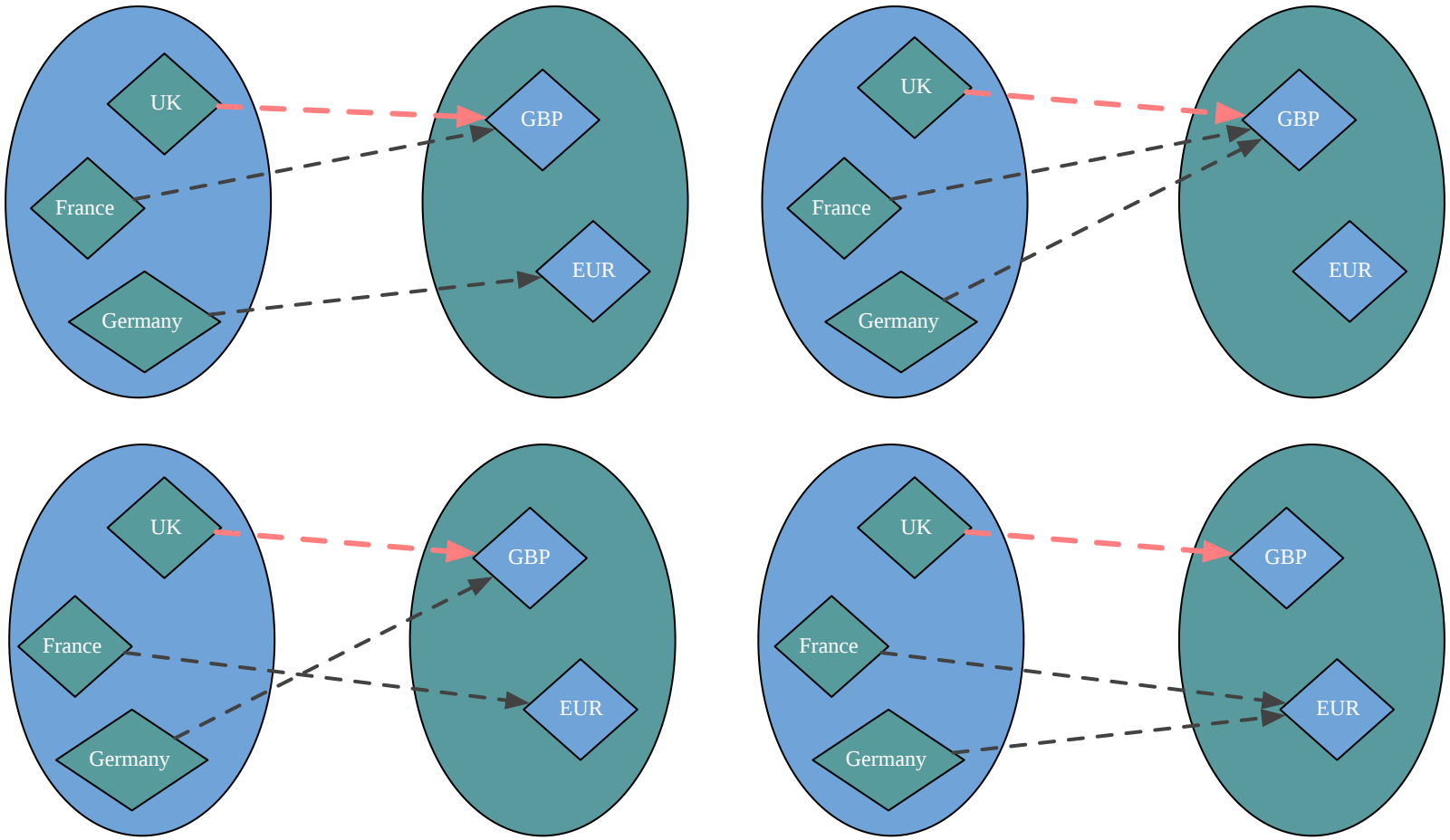
|3 => 2|



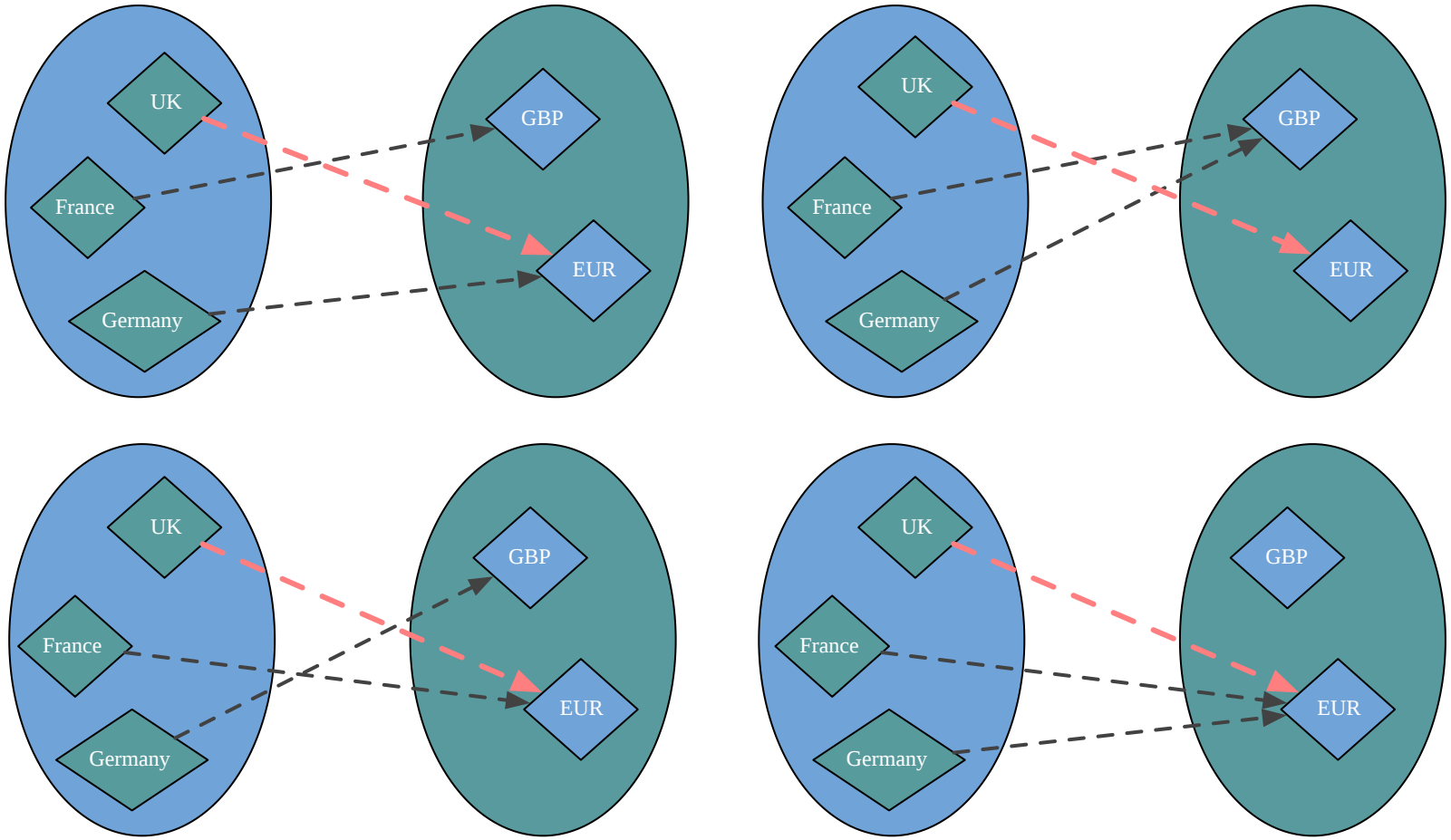
|2 => 2| = 4


France
Germany
GBP
EUR
France
Germany
GBP
EUR
France
Germany
GBP
EUR
France
Germany
GBP
EUR

|3 => 2|


UK
France
Germany
GBP
EUR
UK
France
Germany
GBP
EUR
UK
France
Germany
GBP
EUR
UK
France
Germany
GBP
EUR

|3 => 2| = |2 => 2| + |2 => 2|

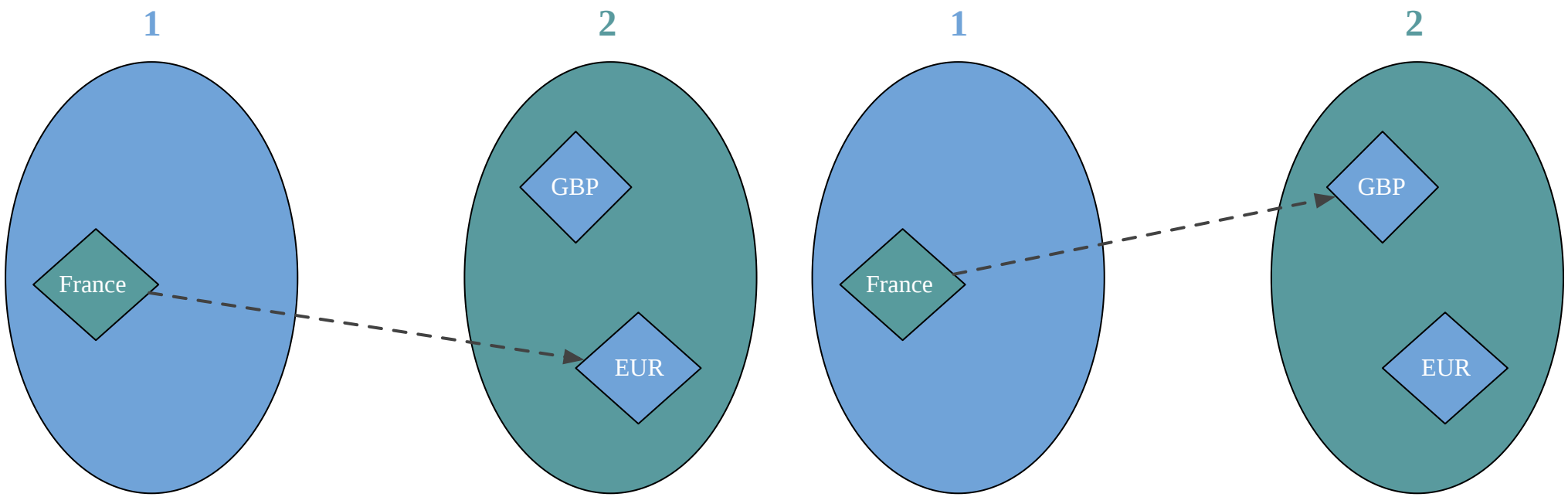

UK
France
Germany
GBP
EUR
UK
France
Germany
GBP
EUR
UK
France
Germany
GBP
EUR
UK
France
Germany
GBP
EUR

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

$$|3 => 2| = 2 * |2 => 2|$$

$$= 2 * 2 * |1 => 2|$$

$$|1 => 2| = 2$$



$$|3 => 2| = 2 * |2 => 2|$$

$$= 2 * 2 * |1 => 2|$$

$$= 2 * 2 * 2$$

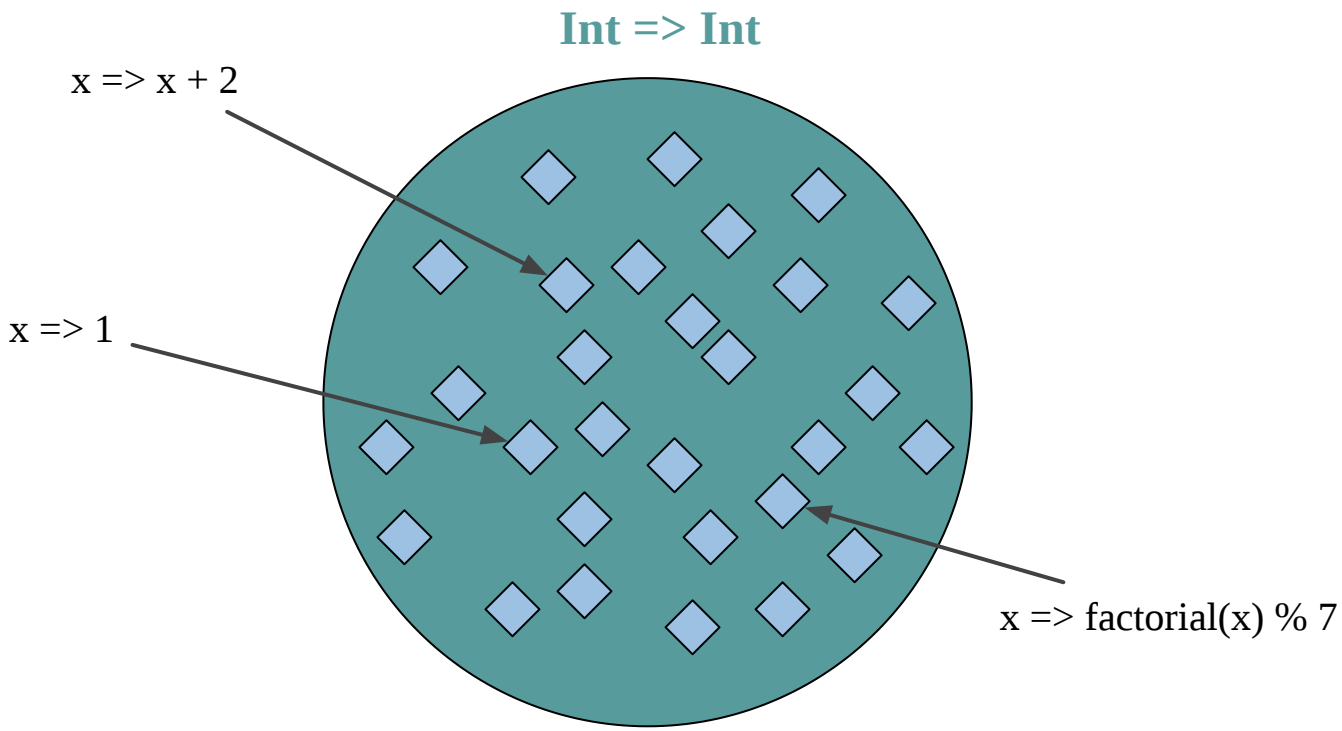
$$
\begin{aligned}
|3 \Rightarrow 2| &= 2 * |2 \Rightarrow 2| \\
&= 2 * 2 * |1 \Rightarrow 2| \\
&= 2 * 2 * 2 \\
&= 2 \wedge 3
\end{aligned}
$$

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

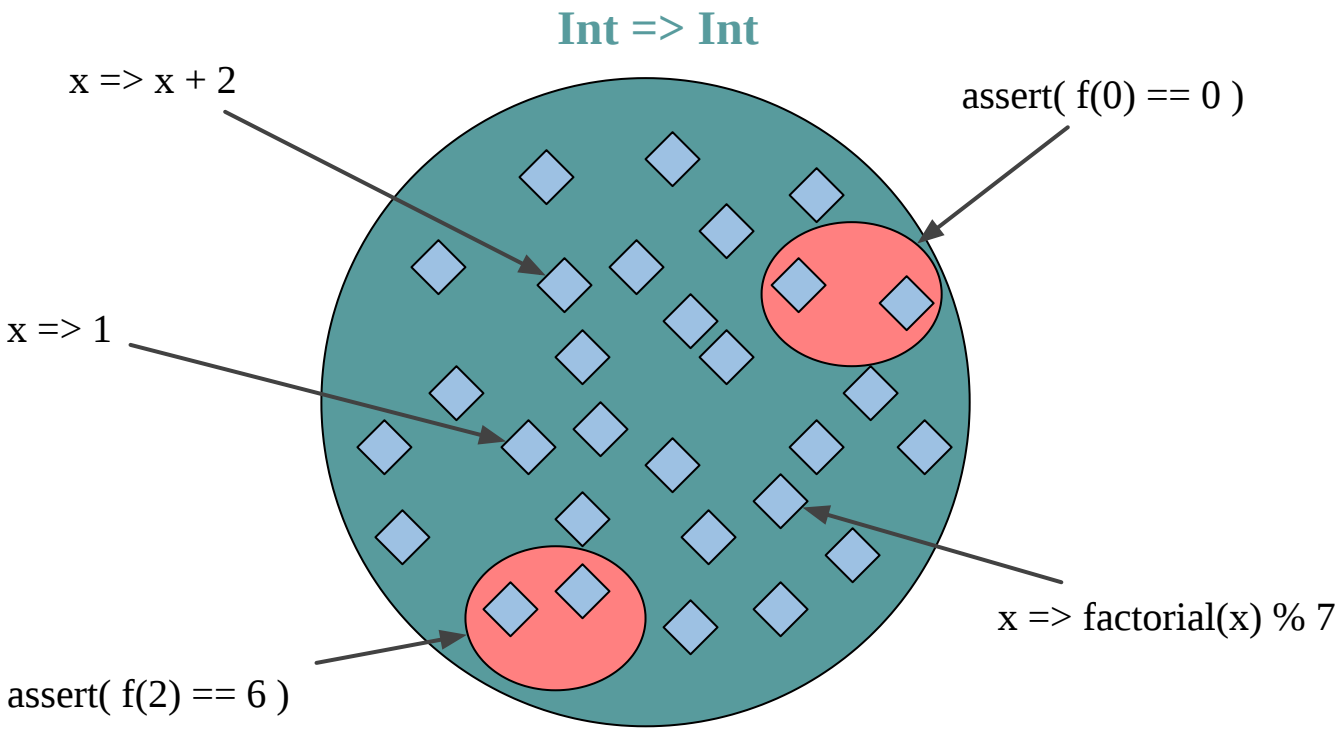
Exercise 4: Parametricity

`exercises.types.TypeExercises.scala`

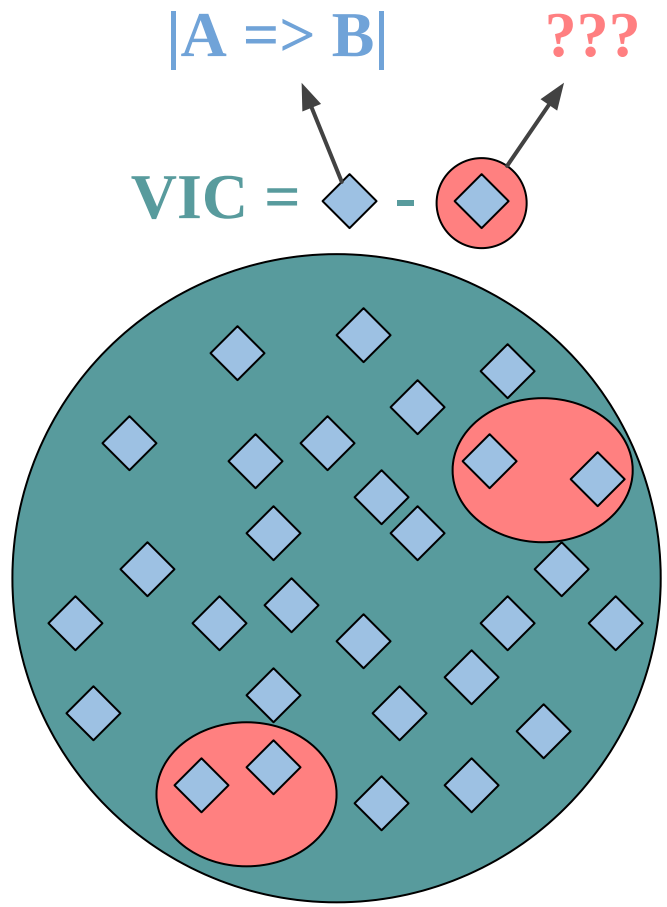
Functions are sets!



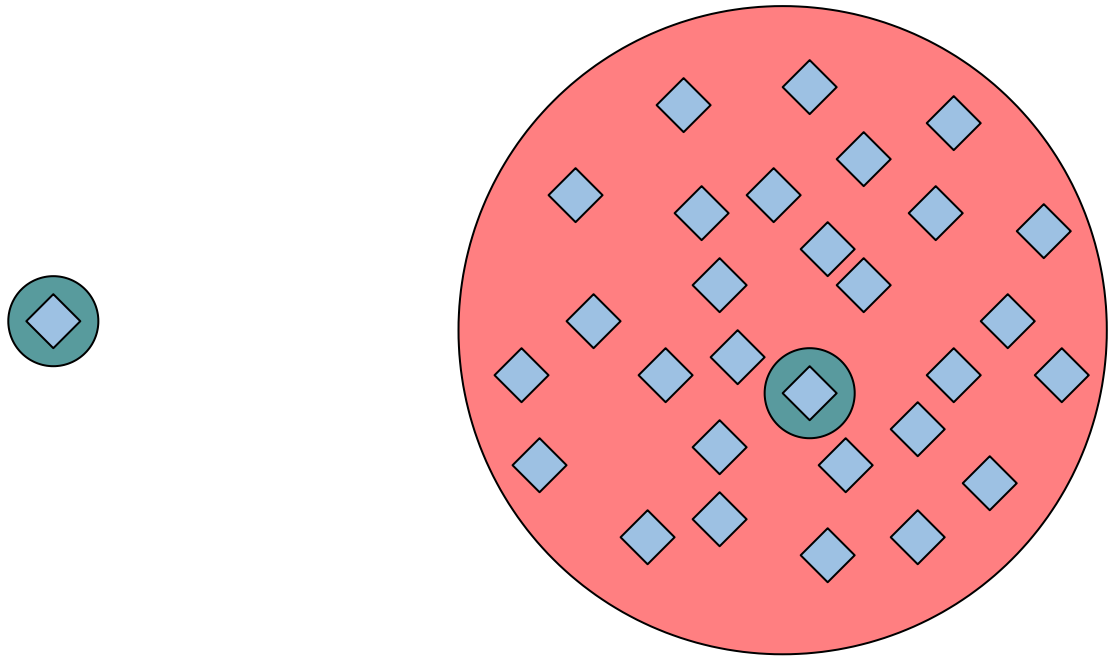
Unit tests



Valid Implementation Count (VIC)


|A => B|
???
VIC = -

VIC(f) = 1



Exercise 5: Tests

`exercises.types.TypeExercises.scala`

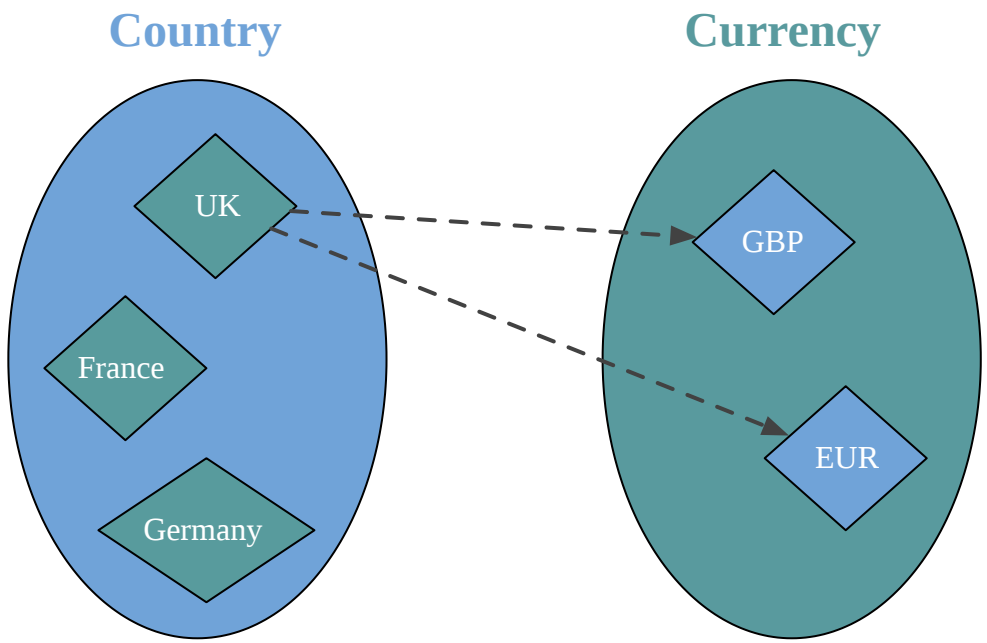
Unit Test

```
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 = ???
```

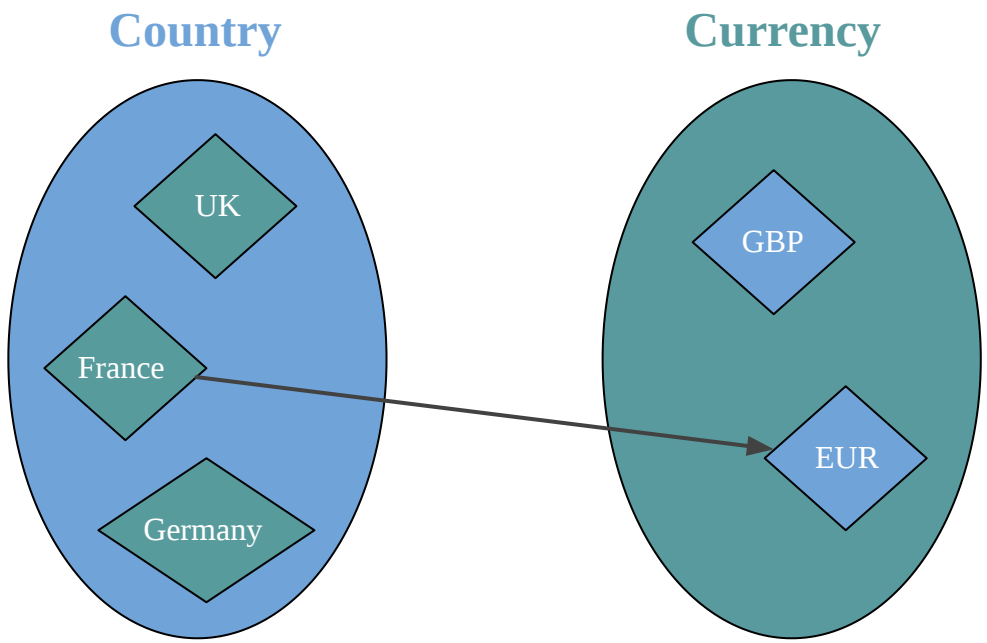
such as

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

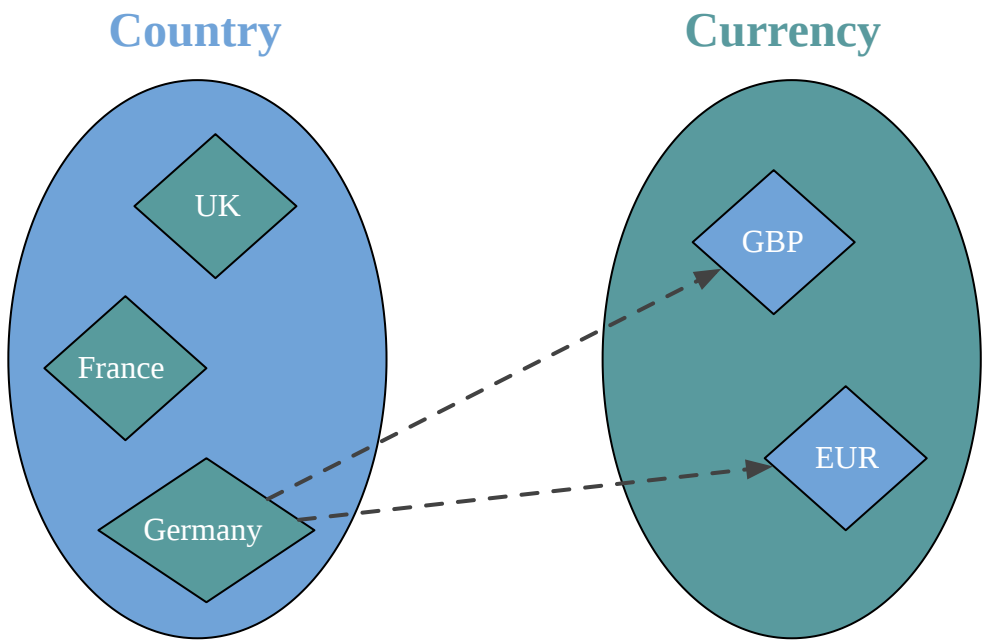
VIC(getCurrency) = 2 * ...



VIC(getCurrency) = 2 * 1 * ...



VIC(getCurrency) = 2 * 1 * 2



$$VIC(f: A => B) = |B| ^ (|A| - n)$$

where n is the number of unit tests

Exercise 6: Type Algebra

`exercises.types.TypeExercises.scala`

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	⊥
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 matters
- 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

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"
}
```

```
inc(5)
// res7: Any = 6
inc(10.3)
// res8: Any = 11.3
inc('c')
// res9: Any = "c1"
```

```
inc(java.time.Instant.ofEpochMilli(0))
// scala.MatchError: 1970-01-01T00:00:00Z (of class java.time.Instant)
//     at repl.Session$App6.inc(4-Type.html:218)
//     at repl.Session$App6$$anonfun$11.apply(4-Type.html:242)
```

$$VIC(Any => Any) = |Any| ^ (|Any| - n)$$

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