









# Scala в законе

Свойства, абстракции, законы на практике

# Кто я

-  .since(2017)
- (  ,  )
- (  ,  ,  )



Алексей Троицкий



[road21/talks](https://github.com/road21/talks)



[@road21](https://t.me/@road21)

# О чем речь

```
trait AbstractAbstraction[A]:  
  
  def abstractOperation(arg1: A, arg2: A ... ): A  
  ...  
  
  def absractValue: A
```

# О чем речь

```
trait AbstractAbstraction[A]:  
  
  // Abstract operation law:  
  //   forall arg1, arg2 ... : A. ... abstractOperation(arg1, arg2 ... ) ...  
  def abstractOperation(arg1: A, arg2: A ... ): A  
  
  ...  
  
  // Abstract value law:  
  //   forall a: A. ... abstractOperation( ... abstractValue ... ) ...  
  def abstractValue: A
```

# О чем речь

```
trait AbstractAbstraction[A]:  
  
  // Abstract operation law:  
  //   forall arg1, arg2 ... : A. ... abstractOperation(arg1, arg2 ... ) ...  
  def abstractOperation(arg1: A, arg2: A ... ): A  
  
  ...  
  
  // Abstract value law:  
  //   forall a: A. ... abstractOperation( ... abstractValue ... ) ...  
  def abstractValue: A
```

Откуда взялись эти законы, какой в них смысл?

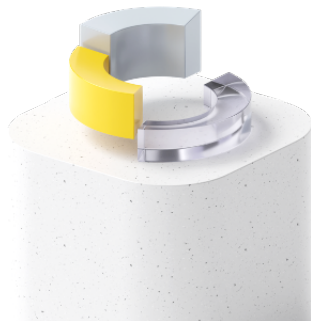
Надо ли их проверять?

А как их вообще проверять?

Где вэлью, Lawfulski?

# План

- Что и зачем: свойства, абстракции, законы
- Проверка свойств и законов
- Другие примеры использования



# Задача

У нас есть:

```
class JsonCache:  
    def put(k: UUID, json: Json): Unit = ...  
    def get(k: UUID): Option[Json] = ...
```

# Задача

У нас есть:

```
class JsonCache:
  def put(k: UUID, json: Json): Unit = ...
  def get(k: UUID): Option[Json] = ...
```

```
forall {
  (cache: JsonCache,
   id: UUID,
   json: Json) =>
  { cache.put(x, json)
    cache.get(x) } = Some(json)
}
```



# Задача

У нас есть:

```
class JsonCache:  
  def put(k: UUID, json: Json): Unit = ...  
  def get(k: UUID): Option[Json] = ...
```

```
forall {  
  (cache: JsonCache,  
   id: UUID,  
   json: Json) =>  
    { cache.put(x, json)  
      cache.get(x) } = Some(json)  
}
```

Хотим кэш для различных сущностей (с UUID идентификаторами):

```
case class User(id: UUID, name: String)  
  
class UserCache(jcache: JsonCache):  
  def put(user: User): Unit = ???  
  def get(id: UUID): Option[User] = ???
```

```
forall {  
  (cache: UserCache, user: User) =>  
    { cache.put(user)  
      cache.get(user.id)  
    } = Some(user)  
}
```

# Json

```
enum Json:  
  case JNull  
  case JBoolean(value: Boolean)  
  case JNumber(value: Long)  
  case JString(value: String)  
  case JArray(value: Vector[Json])  
  case JObject(  
    value: Map[String, Json]  
  )
```

# Json

```
enum Json:  
  case JNull  
  case JBoolean(value: Boolean)  
  case JNumber(value: Long)  
  case JString(value: String)  
  case JArray(value: Vector[Json])  
  case JObject(  
    value: Map[String, Json]  
  )
```

```
case class User(  
  id: UUID,  
  name: String  
)  
  
case class Offer(  
  id: UUID,  
  typ: OfferType  
)
```

# Json

```
enum Json:  
  case JNull  
  case JBoolean(value: Boolean)  
  case JNumber(value: Long)  
  case JString(value: String)  
  case JArray(value: Vector[Json])  
  case JObject(  
    value: Map[String, Json]  
  )
```

```
      write  
    <-----  
    ----->  
      read  
  
      write  
    <-----  
    ----->  
      read
```

```
case class User(  
  id: UUID,  
  name: String  
)  
  
case class Offer(  
  id: UUID,  
  typ: OfferType  
)
```

# EntityCache

```
class EntityCache[A](  
  write: A ⇒ Json,  
  read: Json ⇒ Option[A],  
  key: A ⇒ UUID,  
  cache: JsonCache  
):  
  def put(a: A): Unit =  
    cache.put(key(a), write(a))  
  
  def get(id: UUID): Option[A] =  
    cache.get(id).flatMap(read)
```

# EntityCache

```
class EntityCache[A](
  write: A ⇒ Json,
  read: Json ⇒ Option[A],
  key: A ⇒ UUID,
  cache: JsonCache
):
  def put(a: A): Unit =
    cache.put(key(a), write(a))

  def get(id: UUID): Option[A] =
    cache.get(id).flatMap(read)

//-----

class UserCache(cache: JsonCache)
  extends EntityCache(
    writeUser, readUser, _.id, cache
  )

class OfferCache(cache: JsonCache)
  extends EntityCache(
    writeOffer, readOffer, _.id, cache
  )
```

# EntityCache

```
class EntityCache[A](  
  write: A ⇒ Json,  
  read: Json ⇒ Option[A],  
  key: A ⇒ UUID,  
  cache: JsonCache  
):  
  def put(a: A): Unit =  
    cache.put(key(a), write(a))  
  
  def get(id: UUID): Option[A] =  
    cache.get(id).flatMap(read)
```

```
//-----
```

```
class UserCache(cache: JsonCache)  
  extends EntityCache(  
    writeUser, readUser, _.id, cache  
  )
```

```
class OfferCache(cache: JsonCache)  
  extends EntityCache(  
    writeOffer, readOffer, _.id, cache  
  )
```

```
forall { (uCache: UserCache, user: User) ⇒  
  { uCache.put(user)  
    uCache.get(user.id) } = Some(user) }
```

# EntityCache

```
class EntityCache[A](
  write: A ⇒ Json,
  read: Json ⇒ Option[A],
  key: A ⇒ UUID,
  cache: JsonCache
):
  def put(a: A): Unit =
    cache.put(key(a), write(a))

  def get(id: UUID): Option[A] =
    cache.get(id).flatMap(read)

//-----

class UserCache(cache: JsonCache)
  extends EntityCache(
    writeUser, readUser, _.id, cache
  )

class OfferCache(cache: JsonCache)
  extends EntityCache(
    writeOffer, readOffer, _.id, cache
  )
```

```
forall { (uCache: UserCache, user: User) ⇒
  { uCache.put(user)
    uCache.get(user.id) } = Some(user) }
```



```
forall { (cache: JsonCache, user: User) ⇒ {
  cache.put(user.id, writeUser(user))
  cache.get(user.id).flatMap(readUser)
} = Some(user) }
```



# EntityCache

```
class EntityCache[A](  
  write: A ⇒ Json,  
  read: Json ⇒ Option[A],  
  key: A ⇒ UUID,  
  cache: JsonCache  
):  
  def put(a: A): Unit =  
    cache.put(key(a), write(a))  
  
  def get(id: UUID): Option[A] =  
    cache.get(id).flatMap(read)  
  
//-----  
  
class UserCache(cache: JsonCache)  
  extends EntityCache(  
    writeUser, readUser, _.id, cache  
  )  
  
class OfferCache(cache: JsonCache)  
  extends EntityCache(  
    writeOffer, readOffer, _.id, cache  
  )
```

```
forall { (uCache: UserCache, user: User) ⇒  
  { uCache.put(user)  
    uCache.get(user.id) } = Some(user) }
```



```
forall { (cache: JsonCache, user: User) ⇒ {  
  cache.put(user.id, writeUser(user))  
  cache.get(user.id)  
}.flatMap(readUser) = Some(user) }
```

# EntityCache

```
class EntityCache[A](
  write: A ⇒ Json,
  read: Json ⇒ Option[A],
  key: A ⇒ UUID,
  cache: JsonCache
):
  def put(a: A): Unit =
    cache.put(key(a), write(a))

  def get(id: UUID): Option[A] =
    cache.get(id).flatMap(read)

//-----

class UserCache(cache: JsonCache)
  extends EntityCache(
    writeUser, readUser, _.id, cache
  )

class OfferCache(cache: JsonCache)
  extends EntityCache(
    writeOffer, readOffer, _.id, cache
  )
```

```
forall { (uCache: UserCache, user: User) ⇒
  { uCache.put(user)
    uCache.get(user.id) } = Some(user) }
```



```
forall { (cache: JsonCache, user: User) ⇒ {
  cache.put(user.id, writeUser(user))
  cache.get(user.id)
}.flatMap(readUser) = Some(user) }
```



```
forall { (user: User) ⇒
  Some(writeUser(user))
  .flatMap(readUser) = Some(user) }
```

# EntityCache

```
class EntityCache[A](
  write: A ⇒ Json,
  read: Json ⇒ Option[A],
  key: A ⇒ UUID,
  cache: JsonCache
):
  def put(a: A): Unit =
    cache.put(key(a), write(a))

  def get(id: UUID): Option[A] =
    cache.get(id).flatMap(read)

//-----

class UserCache(cache: JsonCache)
  extends EntityCache(
    writeUser, readUser, _.id, cache
  )

class OfferCache(cache: JsonCache)
  extends EntityCache(
    writeOffer, readOffer, _.id, cache
  )
```

```
forall { (uCache: UserCache, user: User) ⇒
  { uCache.put(user)
    uCache.get(user.id) } = Some(user) }
```



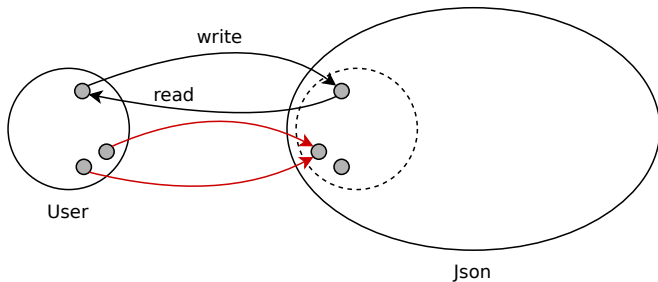
```
forall { (cache: JsonCache, user: User) ⇒ {
  cache.put(user.id, writeUser(user))
  cache.get(user.id)
}.flatMap(readUser) = Some(user) }
```



```
forall { (user: User) ⇒
  readUser(writeUser(user)) = Some(user)
}
```

# write, read

```
forall { (user: User) ⇒  
  readUser(writeUser(user)) = Some(user)  
}
```



# Абстракция

```
trait JsonCodec[A]:  
  /**  
    * forall { (a: A) =>  
    *   write(a).flatMap(read) == Some(a)  
    * }  
    */  
  def write(a: A): Json  
  def read(json: Json): Option[A]
```

# Абстракция

```
trait JsonCodec[A]:  
  /**  
    * forall { (a: A) =>  
    *   write(a).flatMap(read) == Some(a)  
    * }  
    */  
  def write(a: A): Json  
  def read(json: Json): Option[A]
```

```
given JsonCodec[String] with ...  
given JsonCodec[UUID]   with ...  
given JsonCodec[User]   with ...  
given JsonCodec[Offer]  with ...
```

# Абстракция

```
trait JsonCodec[A]:  
  /**  
    * forall { (a: A) =>  
    *   write(a).flatMap(read) = Some(a)  
    * }  
    */  
  def write(a: A): Json  
  def read(json: Json): Option[A]
```

```
given JsonCodec[String] with ...  
given JsonCodec[UUID]   with ...  
given JsonCodec[User]   with ...  
given JsonCodec[Offer]  with ...
```

## Закон (класса типов)

Закон — свойство, которым **обязан** обладать каждый инстанс класса типов

# EntityCache

```
/**
 * forall A: Type, cache: EntityCache[A], a: A.
 *   { cache.put(a) ; cache.get(cache.key(a)) } = Some(a)
 */
case class EntityCache[A](key: A ⇒ UUID, cache: JsonCache)(using codec: JsonCodec[A]):
  def get(id: UUID): Option[A] =
    cache.get(id).flatMap(codec.read)

  def put(a: A): Unit =
    cache.put(key(a), codec.write(a))
```



# Итоги

- Законы помогают писать безопасный обобщенный код, собирая требуемые **ограничения** в одном месте

# Итоги

- Законы помогают писать безопасный обобщенный код, собирая требуемые **ограничения** в одном месте
- Законы обычно не проверяются компилятором, проверка остается на совести разработчика

# Итоги

- Законы помогают писать безопасный обобщенный код, собирая требуемые **ограничения** в одном месте
- Законы обычно не проверяются компилятором, проверка остается на совести разработчика
- В случае, если подставить инстанс класса типов, не удовлетворяющий закону, то целевое требование может не выполняться

# Итоги

- Законы помогают писать безопасный обобщенный код, собирая требуемые **ограничения** в одном месте
- Законы обычно не проверяются компилятором, проверка остается на совести разработчика
- В случае, если подставить инстанс класса типов, не удовлетворяющий закону, то целевое требование может не выполняться



“компилируется — значит работает” не работает\*

\* по крайней мере в Scala\*\*

\*\* по крайней мере в cats

# Задача

Вернуть топ-3 пользователей, отсортированных по возрасту, пользователей без возраста сортировать по имени.

```
case class User(  
  // ...  
  name: String,  
  age: Option[Int]  
  // ...  
)  
  
def top3Users(users: Vector[User]): Vector[User] = ???
```

# top3

```
def top3Users(elems: Vector[User]): Vector[User] =  
  if (elems.size > 3)  
    val (first3, tail) = elems.splitAt(3)  
    tail.foldLeft(first3.sorted) {  
      case (acc, u) =>  
        if (u ≤ acc(0)) acc  
        else (acc.tail :+ u).sorted  
    }.toVector  
  else elems.sorted
```

# top3

```
import scala.math.Ordering.Implicits.infixOrderingOps

def top3Users(elems: Vector[User]): Vector[User] =
  if (elems.size > 3)
    val (first3, tail) = elems.splitAt(3)
    tail.foldLeft(first3.sorted) {
      case (acc, u) =>
        if (u ≤ acc(0)) acc
        else (acc.tail :+ u).sorted
    }.toVector
  else elems.sorted

given Ordering[User] with
  def compare(x: User, y: User): Int =
    (x, y) match
      case (User(_, Some(xAge)), User(_, Some(yAge))) =>
        summon[Ordering[Int]].compare(xAge, yAge)
      case _ =>
        summon[Ordering[String]].compare(x.name, y.name)
```

# top3

```
import scala.math.Ordering.Implicits.infixOrderingOps

def top3Users(users: Vector[User]): Vector[User] = top3(users)

def top3[A: Ordering](elems: Vector[A]): Vector[A] =
  if (elems.size > 3)
    val (first3, tail) = elems.splitAt(3)
    tail.foldLeft(first3.sorted) {
      case (acc, u) =>
        if (u ≤ acc(0)) acc
        else (acc.tail :+ u).sorted
    }.toVector
  else elems.sorted

given Ordering[User] with
  def compare(x: User, y: User): Int =
    (x, y) match
      case (User(_, Some(xAge)), User(_, Some(yAge))) =>
        summon[Ordering[Int]].compare(xAge, yAge)
      case _ =>
        summon[Ordering[String]].compare(x.name, y.name)
```



# Spec

```
class Top3UsersSpec extends munit.FunSuite:
  test("top3Users works"):
    val turing = User("Alan", Some(41))
    val hughes = User("John", None)
    val curry = User("Haskell", Some(81))
    val maclane = User("Saunders", Some(95))
    val yoneda = User("Nobuo", Some(66))
    val pierce = User("Benjamin", None)

    assertEquals(
      top3Users(Vector(maclane, yoneda, curry, hughes, turing, pierce)),
      Vector(yoneda, curry, maclane)
    )
```

# Spec

```
class Top3UsersSpec extends munit.FunSuite:
  test("top3Users works"):
    val turing = User("Alan", Some(41))
    val hughes = User("John", None)
    val curry = User("Haskell", Some(81))
    val maclane = User("Saunders", Some(95))
    val yoneda = User("Nobuo", Some(66))
    val pierce = User("Benjamin", None)

    assertEquals(
      top3Users(Vector(maclane, yoneda, curry, hughes, turing, pierce)),
      Vector(yoneda, curry, maclane)
    )
```

**> test**

```
scala_in_law_md$Scope$Top3Spec:
+ top3 works 0.017s
```

# Spec

```
class Top3UsersSpec extends munit.FunSuite:
  test("top3Users works"):
    val turing = User("Alan", Some(41))
    val hughes = User("John", None)
    val curry = User("Haskell", Some(81))
    val macLane = User("Saunders", Some(95))
    val yoneda = User("Nobuo", Some(66))
    val pierce = User("Benjamin", None)

    assertEquals(
      top3Users(Vector(macLane, yoneda, curry, hughes, turing, pierce)),
      Vector(yoneda, curry, macLane)
    )
```

**> test**

```
scala_in_law_md$Scope$Top3Spec:
+ top3 works 0.017s
```

ОДИН

конкретный

ВХОД

# Property-based Testing

- Проверка **свойств** программы

# Property-based Testing

- Проверка **свойств** программы
- С помощью рандомизированной генерации входных значений

# Property-based Testing

- Проверка **свойств** программы
- С помощью рандомизированной генерации входных значений
- Уменьшение контрпримеров с помощью shrinking-a

# Property-based Testing

- Проверка **свойств** программы
- С помощью рандомизированной генерации входных значений
- Уменьшение контрпримеров с помощью shrinking-a
- ScalaCheck

# Генератор

```
import org.scalacheck.{Arbitrary, Gen}

given Arbitrary[User] =
  Arbitrary(
    for
      name <- Gen.alphaStr
      age  <- Gen.oneOf(
        Gen.const(None),
        Gen.choose(0, 200).map(Some(_))
      )
    yield User(name, age)
  )
```



# Spec

```
class Top3UsersPSpec extends munit.ScalaCheckSuite:  
  property("???"):  
    forAll { (users: Vector[User]) =>  
      top3Users(users) = ???  
    }
```

# Spec

```
class Top3UsersPSpec extends munit.ScalaCheckSuite:  
  property("top3Users works as expected"):  
    forAll { (users: Vector[User]) =>  
      top3Users(users) = expected(users)  
    }  
  
  def expected(users: Vector[User]): Vector[User] = ???
```

# СВОЙСТВО

```
top3(Vector(2, 7, 5, 9, 8, 1, 6)) // (7, 8, 9)  
// 1) размеры результата (3)  
// 2) отсортированность ( $7 \leq 8 \leq 9$ )  
// 3) все остальное меньше ( $2, 5, 1, 6 \leq 7$ )
```

# СВОЙСТВО

```
top3(Vector(2, 7, 5, 9, 8, 1, 6)) // (7, 8, 9)
// 1) размеры результата (3)
// 2) отсортированность ( $7 \leq 8 \leq 9$ )
// 3) все остальное меньше ( $2, 5, 1, 6 \leq 7$ )
```

```
class Top3UsersPSpec extends munit.ScalaCheckSuite:
  property("top3Users result is greatest"):
    forAll { (users: Vector[User]) =>
      val top = top3Users(users)
      val checkSize = if (users.size ≥ 3) top.size = 3
                      else top.size = users.size
      checkSize && { top match
        case h +: t => users.filterNot(top.contains).forall(_ ≤ h) && top.isSorted
        case _      => true
      }
    }
```

# СВОЙСТВО

```
top3(Vector(2, 7, 5, 9, 8, 1, 6)) // (7, 8, 9)
// 1) размеры результата (3)
// 2) отсортированность ( $7 \leq 8 \leq 9$ )
// 3) все остальное меньше ( $2, 5, 1, 6 \leq 7$ )
```

```
class Top3UsersPSpec extends munit.ScalaCheckSuite:
  property("top3Users result is greatest"):
    forAll { (users: Vector[User]) =>
      val top = top3Users(users)
      val checkSize = if (users.size ≥ 3) top.size = 3
                      else top.size = users.size
      checkSize && { top match
        case h +: t => users.filterNot(top.contains).forall(_ ≤ h) && top.isSorted
        case _      => true
      }
    }
```

**> test**

```
⇒ X scala_in_law_md$Scope$Top3UsersPSpec.top3 result is greatest
> ARG_0: Vector(User(ozMaPHW,Some(161)), User(ANSVeJB,Some(161)), User(Msti,None),
>   User(nARSH,None), User(rU,Some(187)))
```

# Проблема

```
def top3Users(users: Vector[User]): Vector[User] = top3(users)
def top3[A: Ordering](elems: Vector[A]): Vector[A] = ...

given Ordering[User] with ...
```

# Проблема

```
def top3Users(users: Vector[User]): Vector[User] = top3(users)
def top3[A: Ordering](elems: Vector[A]): Vector[A] = ...

given Ordering[User] with ...
```

## Ordering

```
trait Ordering[T] extends Comparator[T] with PartialOrdering[T] with Serializable {
  def compare(x: T, y: T): Int

  // Reflexive Law:
  //   forall a: T. a ≤ a
  // Anti-symmetric:
  //   forall a, b: T. a ≤ b && b ≤ a ⇒ a = b
  // Transitive:
  //   forall a, b, c: T. a ≤ b && b ≤ c ⇒ a ≤ c
  override def ltEq(x: T, y: T): Boolean = compare(x, y) ≤ 0

  override def equiv(x: T, y: T): Boolean = compare(x, y) == 0
}
```

# Spec

```
class UserSpec extends munit.ScalaCheckSuite:
  property("user ordering should be reflexive"):
    forAll { (a: User)  $\Rightarrow$  (a  $\leq$  a) = true }

  property("user ordering should be anti-symmetric"):
    forAll { (a: User, b: User)  $\Rightarrow$ 
      (a  $\leq$  b && b  $\leq$  a)  $\Rightarrow$  (a equiv b)
    }

  property("user ordering should be transitive"):
    forAll { (a: User, b: User, c: User)  $\Rightarrow$ 
      (a  $\leq$  b && b  $\leq$  c)  $\Rightarrow$  a  $\leq$  c
    }
```



# Spec

```
class UserSpec extends munit.ScalaCheckSuite:
  property("user ordering should be reflexive"):
    forAll { (a: User) => (a ≤ a) = true }

  property("user ordering should be anti-symmetric"):
    forAll { (a: User, b: User) =>
      (a ≤ b && b ≤ a) => (a equiv b)
    }

  property("user ordering should be transitive"):
    forAll { (a: User, b: User, c: User) =>
      (a ≤ b && b ≤ c) => a ≤ c
    }
```

## > test

```
scala_in_law_md$Scope$Top3UsersPSpec:
=> X scala_in_law_md$Scope$UserSpec.user ordering should be transitive
```

Falsified after 5 passed tests.

```
> ARG_0: User(UZdYmFdbnuSX,None)
> ARG_1: User(xDiuONvzchzXKsbHbRqCPnN,Some(56))
> ARG_2: User(TzYeTfuIGkTjVHSSkKZcov,Some(90))
```

# Исправляем

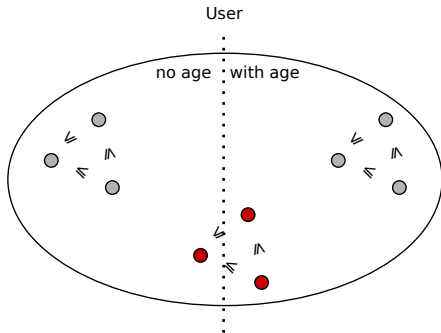
```
given Ordering[User] with
  def compare(x: User, y: User): Int =
    (x, y) match
      case (User(_, Some(xAge)), User(_, Some(yAge))) =>
        summon[Ordering[Int]].compare(xAge, yAge)

      case _ =>
        summon[Ordering[String]].compare(x.name, y.name)
```

# Исправляем

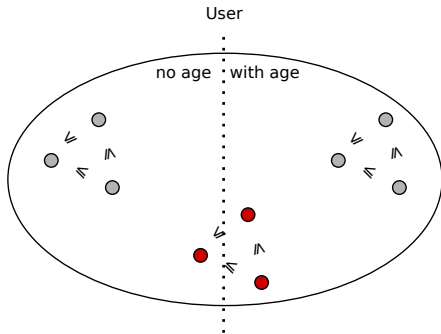
```
given Ordering[User] with
  def compare(x: User, y: User): Int =
    (x, y) match
      case (User(_, Some(xAge)), User(_, Some(yAge))) =>
        summon[Ordering[Int]].compare(xAge, yAge)

      case _ =>
        summon[Ordering[String]].compare(x.name, y.name)
```



# Исправляем

```
given Ordering[User] with
def compare(x: User, y: User): Int =
  (x, y) match
    case (User(_, Some(xAge)), User(_, Some(yAge))) =>
      summon[Ordering[Int]].compare(xAge, yAge)
    case (User(_, Some(_)), _) => 1
    case (_, User(_, Some(_))) => -1
    case _ =>
      summon[Ordering[String]].compare(x.name, y.name)
```



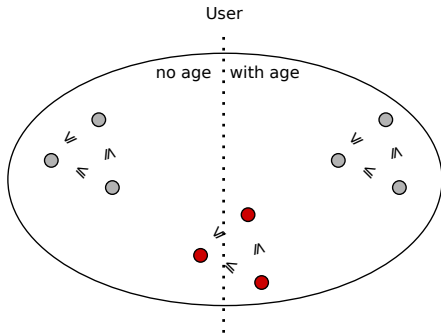
# Исправляем

```
given Ordering[User] with
def compare(x: User, y: User): Int =
  (x, y) match
    case (User(_, Some(xAge)), User(_, Some(yAge))) =>
      summon[Ordering[Int]].compare(xAge, yAge)
    case (User(_, Some(_)), _) => 1
    case (_, User(_, Some(_))) => -1
    case _ =>
      summon[Ordering[String]].compare(x.name, y.name)
```

**> test**

```
... UserSpec:
  + user ordering ...

... Top3UsersPSpec:
  + top3Users result ...
```



# Итоги

- Для проверки законов можно использовать **property-based testing**

# Итоги

- Для проверки законов можно использовать **property-based testing**
- Законы можно писать не только в комментариях, но и в виде `ScalaCheck` свойств, которые:
  - можно переиспользовать
  - не надо придумывать с нуля

# Итоги

- Для проверки законов можно использовать **property-based testing**
- Законы можно писать не только в комментариях, но и в виде `ScalaCheck` свойств, которые:
  - можно переиспользовать
  - не надо придумывать с нуля
- С помощью property-based тестов имеет смысл проверять те (и только те) экземпляры, которые пишутся руками

## (not yet best) practice

Написал кастомный экземпляр класса типов в законах — добавь property-based тест на эти законы



# Ссылочная прозрачность

## Ссылочная прозрачность (referential transparency)

Свойство кусочка кода (метод, блок): можно заменить вызов этого кода на значение, которое он вычисляет, и наоборот, при этом семантика программы не изменится

# Ссылочная прозрачность

## Ссылочная прозрачность (referential transparency)

Свойство кусочка кода (метод, блок): можно заменить вызов этого кода на значение, которое он вычисляет, и наоборот, при этом семантика программы не изменится

```
def rtFunction(a: Int, b: Int): Int =  
  a + b  
  
val compute =  
  val comp1 = rtFunction(2, 5)  
  val comp2 = rtFunction(2, 5)  
  val comp3 = rtFunction(2, 5)  
  
  comp1 + comp2 + comp3
```

# Ссылочная прозрачность

## Ссылочная прозрачность (referential transparency)

Свойство кусочка кода (метод, блок): можно заменить вызов этого кода на значение, которое он вычисляет, и наоборот, при этом семантика программы не изменится

```
def rtFunction(a: Int, b: Int): Int =  
  a + b  
  
val compute =  
  val comp1 = rtFunction(2, 5)  
  val comp2 = rtFunction(2, 5)  
  val comp3 = rtFunction(2, 5)  
  
  comp1 + comp2 + comp3
```



```
def rtFunction(a: Int, b: Int): Int =  
  a + b  
  
val compute =  
  val comp = rtFunction(2, 5)  
  comp + comp + comp
```

# Свойства

Другие свойства программ также можно использовать для оптимизаций!

## List property

```
forall { (l: List[A], f: A => B, g: B => C) =>
  l.map(f).map(g) == l.map(g compose f)
}
```

# СВОЙСТВА

Другие свойства программ также можно использовать для оптимизаций!

## List property

```
forall { (l: List[A], f: A ⇒ B, g: B ⇒ C) ⇒  
  l.map(f).map(g) = l.map(g compose f)  
}
```

```
val list = List(1, 2, 3)  
  
val f: Int ⇒ Int = _ + 1  
val g: Int ⇒ Int = _ * 2  
  
list.map(f).map(g)
```

# СВОЙСТВА

Другие свойства программ также можно использовать для оптимизаций!

## List property

```
forall { (l: List[A], f: A ⇒ B, g: B ⇒ C) ⇒  
  l.map(f).map(g) = l.map(g compose f)  
}
```

```
val list = List(1, 2, 3)  
  
val f: Int ⇒ Int = _ + 1  
val g: Int ⇒ Int = _ * 2  
  
list.map(f).map(g)
```



```
val list = List(1, 2, 3)  
  
val f: Int ⇒ Int = _ + 1  
val g: Int ⇒ Int = * 2  
  
list.map(g compose f)
```

# А законы?

## Functor

```
trait Functor[F[_]]:
  /**
   * Identity law:
   * forAll { (fa: F[A]) =>
   *   map(fa, identity) == fa
   * }
   *
   * Composition law:
   * forAll { (fa: F[A], f: A => B, g: B => C) =>
   *   map(map(fa)(f), g) == map(fa, (g compose f))
   * }
   */
  def map[A](fa: F[A])(f: A => B): F[B]
```

# А законы?

## Functor

```
trait Functor[F[_]]:
  /**
   * Identity law:
   * forAll { (fa: F[A]) =>
   *   map(fa, identity) == fa
   * }
   *
   * Composition law:
   * forAll { (fa: F[A], f: A => B, g: B => C) =>
   *   map(map(fa)(f), g) == map(fa, (g compose f))
   * }
   */
  def map[A](fa: F[A])(f: A => B): F[B]
```

```
def fg[F[_]](fa: F[Int])(
  using F: Functor[F]
): F[Int] =
  F.map(F.map(fa, f), g)
```



# А законы?

## Functor

```
trait Functor[F[_]]:
  /**
   * Identity law:
   * forAll { (fa: F[A]) =>
   *   map(fa, identity) == fa
   * }
   *
   * Composition law:
   * forAll { (fa: F[A], f: A => B, g: B => C) =>
   *   map(map(fa)(f), g) == map(fa, (g compose f))
   * }
   */
  def map[A](fa: F[A])(f: A => B): F[B]
```

```
def fg[F[_]](fa: F[Int])(
  using F: Functor[F]
): F[Int] =
  F.map(F.map(fa, f), g)
```

???



```
def fg[F[_]](fa: F[Int])(
  using F: Functor[F]
): F[Int] =
  F.map(fa, g compose f)
```

# Когда очевидно

```
// Functor identity law
forall { (fa: F[A]) =>
  map(fa, identity) = fa
}
```

# Когда очевидно

```
// Functor identity law
forAll { (fa: F[A]) =>
  map(fa, identity) = fa
}
```

```
// Right identity Monad law
forAll { (fa: F[A]) =>
  flatMap(fa, pure) = fa
}
```

# Когда очевидно

```
// Functor identity law
forall { (fa: F[A]) =>
  map(fa, identity) = fa
}
```

```
// Right identity Monad law
forall { (fa: F[A]) =>
  flatMap(fa, pure) = fa
}
```


```
// Numeric law
forall { (x: A) =>
  abs(x) * signum(x) = x
}
```

# Когда очевидно

```
// Functor identity law
forAll { (fa: F[A]) =>
  map(fa, identity) = fa
}
```

```
// Right identity Monad law
forAll { (fa: F[A]) =>
  flatMap(fa, pure) = fa
}
```

```
// Numeric law
forAll { (x: A) =>
  abs(x) * signum(x) = x
}
```

 Для этих законов в Haskell есть соответствующие rewrite правила

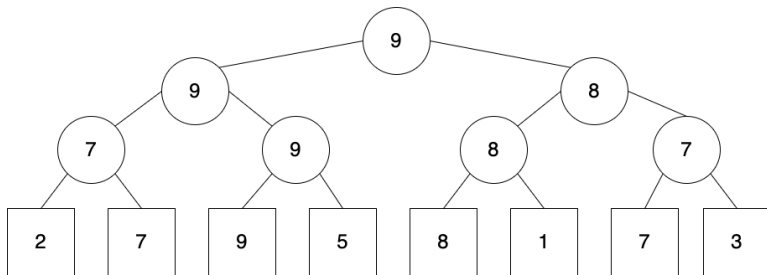
# Максимум на отрезке

**Задача:** дан список чисел, надо уметь эффективно отвечать на запрос  $\max(i, j)$

# Максимум на отрезке

**Задача:** дан список чисел, надо уметь эффективно отвечать на запрос  $\max(i, j)$

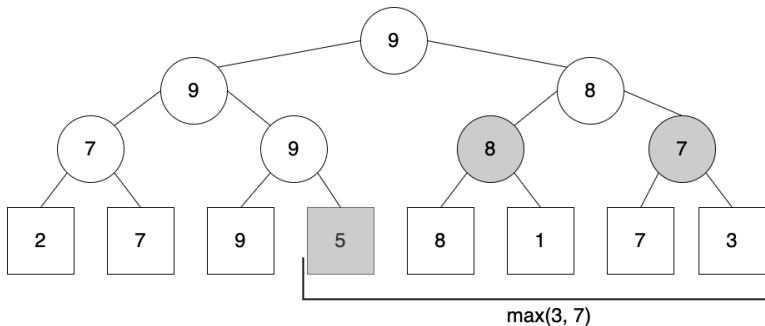
Ассоциативность  $\max$  позволяет считать частичные результаты заранее:



# Максимум на отрезке

**Задача:** дан список чисел, надо уметь эффективно отвечать на запрос  $\max(i, j)$

Ассоциативность  $\max$  позволяет считать частичные результаты заранее:

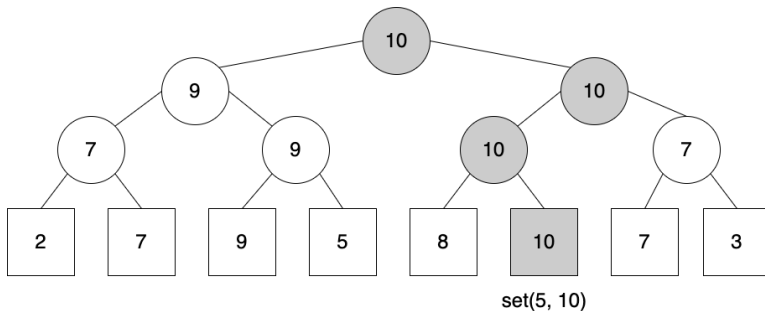




# Максимум на отрезке

**Задача:** дан список чисел, надо уметь эффективно отвечать на запрос  $\max(i, j)$

Ассоциативность  $\max$  позволяет считать частичные результаты заранее:



# SegmentTree

```
trait SegTree[A]:  
  def calc(l: Int, r: Int): A = ...  
  def set(idx: Int, v: A): SegTree[A] = ...  
  
object SegTree:  
  def apply[A: Semigroup](v: Vector[A]): STree[A] = ...
```

# TailCalls

```
enum Tree[A]:  
  case Node(ts: Vector[Tree[A]])  
  case Leaf(v: A)  
  
def map[A, B](tree: Tree[A], f: A ⇒ B): Tree[B] =  
  tree match  
    case Tree.Node(ts) ⇒ Tree.Node(ts.map(map(_, f)))  
    case Tree.Leaf(v)   ⇒ Tree.Leaf(f(v))
```

# TailCalls

```
import scala.util.control.TailCalls

enum Tree[A]:
  case Node(ts: Vector[Tree[A]])
  case Leaf(v: A)

def map[A, B](tree: Tree[A], f: A ⇒ B): Tree[B] =
  def run(in: Tree[A]): TailCalls.TailRec[Tree[B]] = in match
    case Tree.Node(ts) ⇒
      ts.traverse(t ⇒ TailCalls.tailcall(run(t))).map(Tree.Node(_))
    case Tree.Leaf(v) ⇒
      TailCalls.done(Tree.Leaf(f(v)))

  run(tree).result
```

# TailCalls.TailRec

```
case class Done[A](value: A) extends TailRec[A]
case class Call[A](rest: () => TailRec[A]) extends TailRec[A]
case class Cont[A, B](a: TailRec[A], f: A => TailRec[B]) extends TailRec[B]

abstract class TailRec[+A] {
```

```
}
```

# TailCalls.TailRec

```
case class Done[A](value: A) extends TailRec[A]
case class Call[A](rest: () => TailRec[A]) extends TailRec[A]
case class Cont[A, B](a: TailRec[A], f: A => TailRec[B]) extends TailRec[B]

abstract class TailRec[+A] {
  final def flatMap[B](f: A => TailRec[B]): TailRec[B] = this match {
    case Done(a) => Call(() => f(a))
    case c@Call(_) => Cont(c, f)
    // Use the monad associative law to optimize the size of the required stack
    case c: Cont[a1, b1] => Cont(c.a, (x: a1) => c.f(x) flatMap f)
  }
}
```

}

# TailCalls.TailRec

```
case class Done[A](value: A) extends TailRec[A]
case class Call[A](rest: () => TailRec[A]) extends TailRec[A]
case class Cont[A, B](a: TailRec[A], f: A => TailRec[B]) extends TailRec[B]

abstract class TailRec[+A] {
  final def flatMap[B](f: A => TailRec[B]): TailRec[B] = this match {
    case Done(a) => Call(() => f(a))
    case c@Call(_) => Cont(c, f)
    // Use the monad associative law to optimize the size of the required stack
    case c: Cont[a1, b1] => Cont(c.a, (x: a1) => c.f(x) flatMap f)
  }

  @annotation.tailrec final def result: A = this match {
    case Done(a) => a
    case Call(t) => t().result
    case Cont(a, f) => a match {
      case Done(v) => f(v).result
      case Call(t) => t().flatMap(f).result
      case Cont(b, g) => b.flatMap(x => g(x) flatMap f).result
    }
  }
}
```

# Итоги

Законы помогают:

- писать безопасный обобщенный код



# Итоги

Законы помогают:

- писать безопасный обобщенный код
- безопасно “рефакторить” код, получая выигрыш в плане оптимизации

# Итоги

Законы помогают:

- писать безопасный обобщенный код
- безопасно “рефакторить” код, получая выигрыш в плане оптимизации

А на практике:

# Итоги

Законы помогают:

- писать безопасный обобщенный код
- безопасно “рефакторить” код, получая выигрыш в плане оптимизации

А на практике:

- законы можно проверять с помощью property-based тестов

# Итоги

Законы помогают:

- писать безопасный обобщенный код
- безопасно “рефакторить” код, получая выигрыш в плане оптимизации

А на практике:

- законы можно проверять с помощью property-based тестов
- удобно переиспользуя `ScalaCheck` свойства

# Итоги

Законы помогают:

- писать безопасный обобщенный код
- безопасно “рефакторить” код, получая выигрыш в плане оптимизации

А на практике:

- законы можно проверять с помощью property-based тестов
- удобно переиспользуя `ScalaCheck` свойства
- всякий раз когда пишешь инстансы классов типов руками

# Благодарности

- Денис Буздалов
- Иван Лягаев
- Борис Потепун
- Олег Нижников
- Петр и Александра Т.



# Ссылки

- John Hughes — [How to specify it!](#)
- Михаил Чугунков — [Как в Scala переложить JSON: паттерн «codec»](#)
- [Статья](#) про дерево отрезков на хабре
- Rúnar Bjarnason — [Stackless Scala With Free Monads](#)



[github.com/road21/talks](https://github.com/road21/talks)

# Ссылки

- John Hughes — [How to specify it!](#)
- Михаил Чугунков — [Как в Scala переложить JSON: паттерн «codec»](#)
- [Статья](#) про дерево отрезков на хабре
- Rúnar Bjarnason — [Stackless Scala With Free Monads](#)



[github.com/road21/talks](https://github.com/road21/talks)

## Спасибо!

made with love,

scala-cli, pandoc, L<sup>A</sup>T<sub>E</sub>X