

$$S \xrightarrow{f} T$$

$$G \xrightarrow{\varphi} H$$

$$X \xrightarrow{g} Y$$

Criteria4s

Filtros, filtros everywhere

Rafael Fernández Ortiz

Hello folks! 🙋

Rafael Fernández

Formal Method Software Engineer | Mathematician | Scala Developer

Release 1994

Linkedin [rafael-fernandez-ortiz](#)

Github [@rafafrdz](#)

Criteria4s [github.com/rafafrdz/criteria4s](#)

Presentation [github.com/rafafrdz/criteria4s-presentation](#)



Contexto - Inicios

¿Cómo empezó todo?

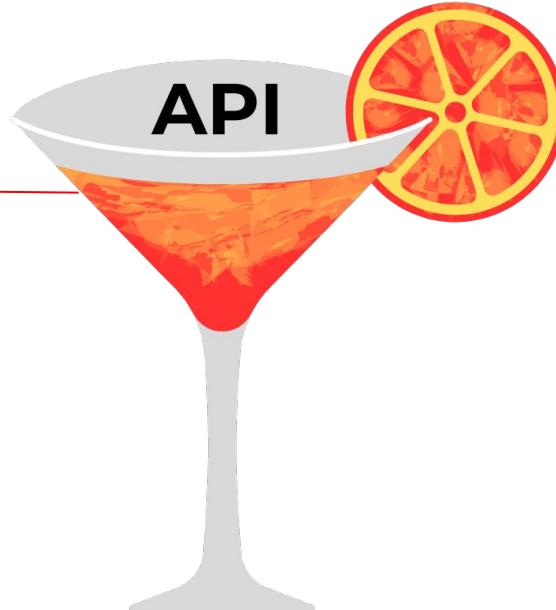
Inquietud por nuevas tecnologías



Usando del Scala

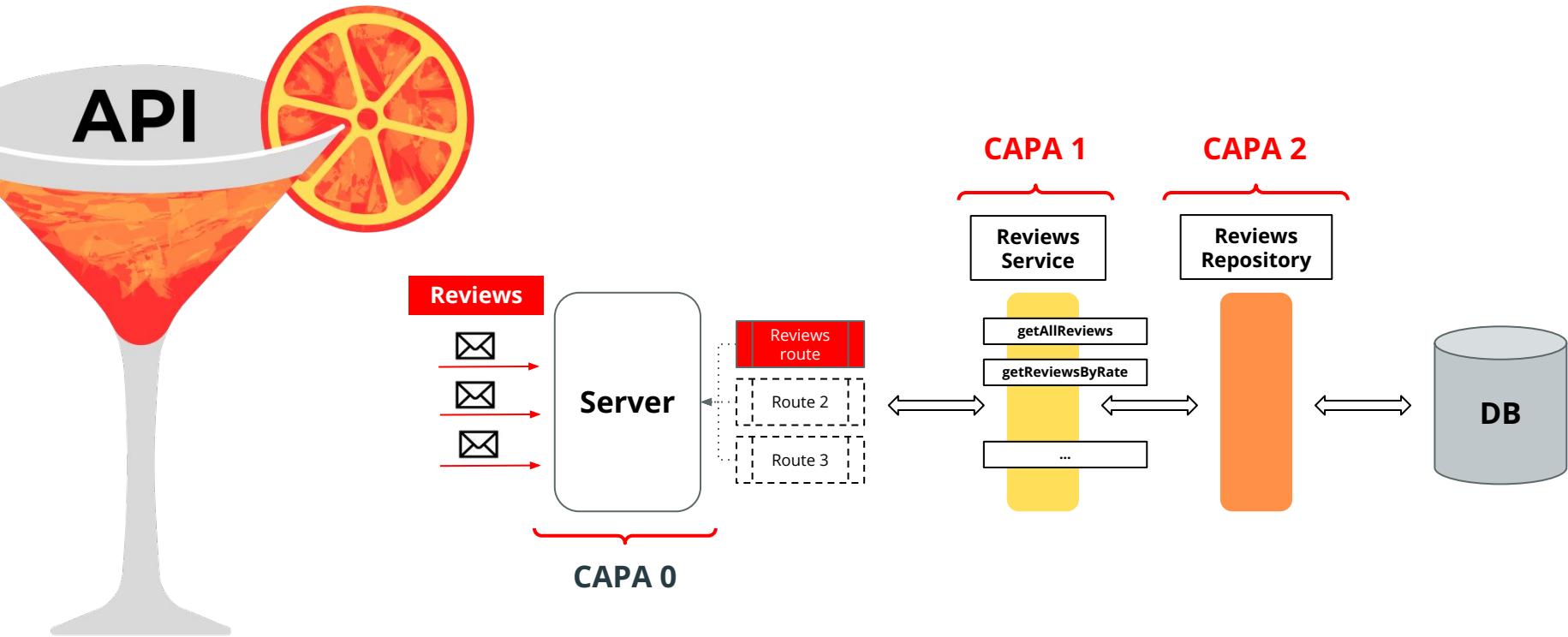


EL RESULTADO; Algo palpable



Contexto - API

¿Cómo funciona? Y ¿Cómo es su estructura?

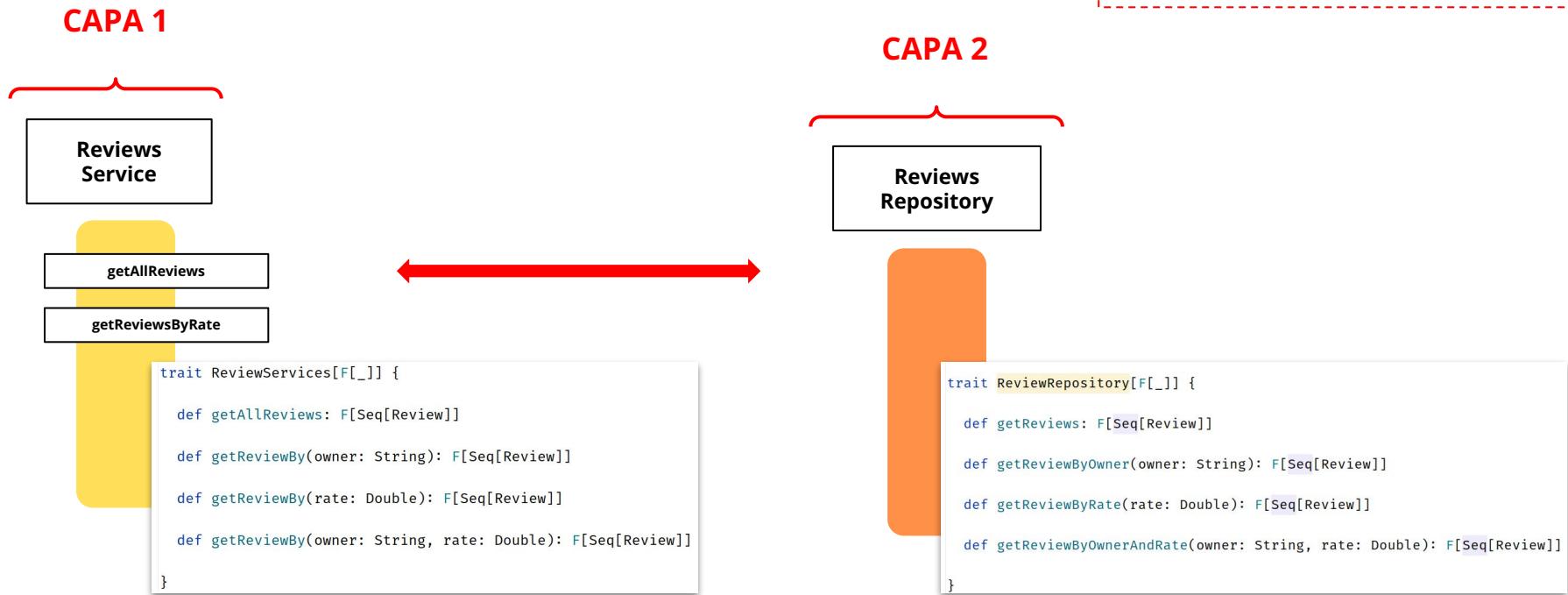


Contexto - Viaje entre capas

Planteamiento problema

¿Qué vemos?

- Mapeo 1-1
- Un criterio = un método en ambas capas

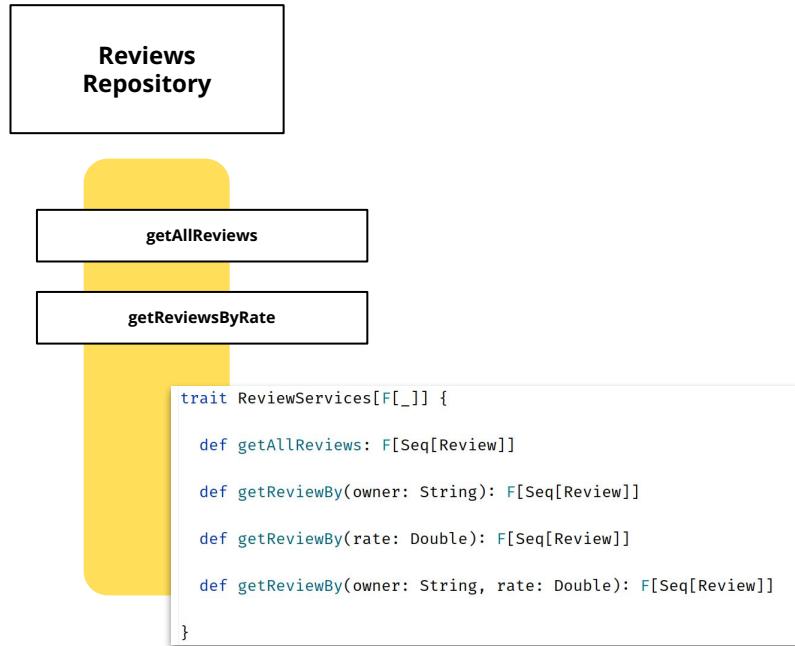


Contexto - Viendo cada capa

Implementación de CAPA 1 = Service

¿Qué vemos?

- Depende de la capa 2 = repository
- Agnóstico a la DB



Ejemplo:

✓

```
/** Generic ReviewServices implementation */  
def build[F[_]: Async: Logger](  
    rep: ReviewRepository_[F]  
) : ReviewServices[F] =  
    new ReviewServices[F] {  
  
        override def getAllReviews: F[Seq[Review]] =  
            Logger[F].info("Getting all reviews") => rep.getReviews  
  
        override def getReviewBy(owner: String): F[Seq[Review]] =  
            Logger[F].info(s"Getting reviews by owner: $owner") =>  
            rep.getReviewByOwner(owner)  
  
        override def getReviewBy(rate: Double): F[Seq[Review]] =  
            Logger[F].info(s"Getting reviews by rate: $rate") =>  
            rep.getReviewByRate(rate)  
  
        override def getReviewBy(owner: String, rate: Double): F[Seq[Review]] =  
            Logger[F].info(s"Getting reviews by owner: $owner and rate: $rate") =>  
            rep.getReviewByOwnerAndRate(owner, rate)  
    }  
}
```

Contexto - Viendo cada capa

Implementación de CAPA 2 = Repository

Reviews
Repository

¿Qué vemos?

- Dependiente a la DB
- Diferentes expresiones de criterios

```
trait ReviewRepository[F[_]] {  
  
    def getReviews: F[Seq[Review]]  
  
    def getReviewByOwner(owner: String): F[Seq[Review]]  
  
    def getReviewByRate(rate: Double): F[Seq[Review]]  
  
    def getReviewByOwnerAndRate(owner: String, rate: Double): F[Seq[Review]]  
  
}
```



Ejemplos:

```
/** MongoDB ReviewRepository implementation */  
def using[F[_]: Async: Logger](mongo: MongoClient): ReviewRepository[F] =  
  new ReviewRepository[F] {  
  
    def db: MongoDatabase = mongo.getDatabase("Airbnb")  
    def collection: MongoCollection[Document] = db.getCollection("Reviews")  
  
    override def getReviewBy(owner: String): F[Seq[Review]] =  
      for {  
        docs ← Async[F].fromFuture(  
          collection.find(Document(s"""{ owner: { $$eq: "$owner" } }""").toFuture().pure[F])  
        reviews = docs.map(documentToReview)  
      } yield reviews  
  
    override def getReviewBy(rate: Double): F[Seq[Review]] =  
      for {  
        docs ← Async[F].fromFuture(  
          collection.find(Document(s"""{ rate: { $$gt: $rate } }""").toFuture().pure[F])  
        reviews = docs.map(documentToReview)  
      } yield reviews  
  
    override def getReviewBy(owner: String, rate: Double): F[Seq[Review]] =  
      for {  
        docs ← Async[F].fromFuture(  
          collection.find(  
            Document(s"""{ $and: [ { owner: { $$eq: "$owner" } }, { rate: { $$gt: $rate } } ] }""").  
            toFuture().pure[F]  
        )  
        reviews = docs.map(documentToReview)  
      } yield reviews
```

```
/** JDBC ReviewRepository implementation */  
def using[F[_]: Async: Logger](jdbc: Connection): ReviewRepository[F] =  
  new ReviewRepository[F] {  
  
    override def getReviewBy(owner: String): F[Seq[Review]] =  
      execute(jdbc, s"SELECT * FROM $Airbnb.$Reviews WHERE owner = '$owner'" ) { rs ⇒  
        resultSetToReview(rs)  
      }  
  
    override def getReviewBy(rate: Double): F[Seq[Review]] =  
      execute(jdbc, s"SELECT * FROM $Airbnb.$Reviews WHERE rate > '$rate'" ) { rs ⇒  
        resultSetToReview(rs)  
      }  
  
    override def getReviewBy(owner: String, rate: Double): F[Seq[Review]] =  
      execute(jdbc, s"SELECT * FROM $Airbnb.$Reviews WHERE (owner = '$owner') AND (rate > '$rate')") { rs ⇒  
        resultSetToReview(rs)  
      }  
  }
```

Contexto - Viendo cada capa

Experimentando con la **CAPA 2 = Repository (II)**

Y si...

```
trait ReviewRepository_[F[_]] {  
  
    def getReviews: F[Seq[Review]]  
  
    def getReviewByOwner(owner: String): F[Seq[Review]]  
  
    def getReviewByRate(rate: Double): F[Seq[Review]]  
  
    def getReviewByOwnerAndRate(owner: String, rate: Double): F[Seq[Review]]  
  
    def getReviewByAddress(address: String): F[Seq[Review]]  
  
    def getReviewByRateAndAddress(rate: Double, address: String): F[Seq[Review]]  
  
    def getReviewByOwnerAndAddress(owner: String, address: String): F[Seq[Review]]  
  
    def getReviewByOwnerRateAndAddress(owner: String, rate: Double, address: String): F[Seq[Review]]  
  
    def getReviewByComment(comment: String): F[Seq[Review]]  
  
    def getReviewByRateAndComment(rate: Double, comment: String): F[Seq[Review]]  
  
    def getReviewByOwnerAndComment(owner: String, comment: String): F[Seq[Review]]  
  
    def getReviewByOwnerRateAndComment(owner: String, rate: Double, comment: String): F[Seq[Review]]  
  
    def getReviewByAddressAndComment(address: String, comment: String): F[Seq[Review]]  
  
}
```



¿Qué hacemos?

- Aumentan los criterios
- Diferentes DB

¿Qué vemos? - RESULTADO

- Producto cartesiano implementaciones
- No escala

Contexto - Viendo cada capa

Experimentando con la **CAPA 2 = Repository (III)**

Y si...

```
trait ReviewRepository_[F[_]] {  
  
    def getReviews: F[Seq[Review]]  
  
    def getReviewByOwner(owner: String): F[Seq[Review]]  
  
    def getReviewByRate(rate: Double): F[Seq[Review]]  
  
    def getReviewByOwnerAndRate(owner: String, rate: Double): F[Seq[Review]]  
  
    def getReviewByAddress(address: String): F[Seq[Review]]  
  
    def getReviewByRateAndAddress(rate: Double, address: String): F[Seq[Review]]  
  
    def getReviewByOwnerAndAddress(owner: String, address: String): F[Seq[Review]]  
  
    def getReviewByOwnerRateAndAddress(owner: String, rate: Double, address: String): F[Seq[Review]]  
  
    def getReviewByComment(comment: String): F[Seq[Review]]  
  
    def getReviewByRateAndComment(rate: Double, comment: String): F[Seq[Review]]  
  
    def getReviewByOwnerAndComment(owner: String, comment: String): F[Seq[Review]]  
  
    def getReviewByOwnerRateAndComment(owner: String, rate: Double, comment: String): F[Seq[Review]]  
  
    def getReviewByAddressAndComment(address: String, comment: String): F[Seq[Review]]  
  
}
```



¿Qué hacemos?

- Generalizar / unificar el criterio
- ¿Cómo se va a expresar?
- ¿Qué forma va a tener? (*String, ADT, etc.*)

```
trait ReviewRepository[F[_]] {  
  
    def getReviews: F[Seq[Review]]  
  
    def getReviewBy(criteria: String): F[Seq[Review]]  
  
}
```

```
trait ReviewRepository[F[_]] {  
  
    def getReviews: F[Seq[Review]]  
  
    def getReviewBy(criteria: Criteria): F[Seq[Review]]  
  
}
```



Planteamiento inicial - ¿Qué es un criterio?

CONCEPTO

Es una expresión que cumple una condición o predicado

Ejemplos:

- **True / False** (*valores*)
- **4 > 0** (*expresión simple*)
- **(x > 0 and x <= 10) or (x > 20)** (*expresión compleja*)
- **`USER_ID` = '07715cee-5d87'** (*expresión DB*)

OPERACIONES

Encontramos:

- Operan con expresiones Booleanas
 - **AND / OR**
- Generan expresiones Booleanas
 - **>** (*mayor que*)
 - **==** (*igual que*)
 - **isNull** (*el valor es un valor null*)



Planteamiento inicial - ¿Qué es un criterio?

REPRESENTACIÓN

Diferentes formas de representar una expresión

- $(x > 0 \ \&\ x \leq 10) \mid\mid (x > 20)$
- $(('x' > 0) \text{ AND } ('x' \leq 10)) \text{ OR } ('x' > 20)$
- {
opt: OR,
left: {
 opt: AND,
 left: { left: x, opt: >, right: 0 },
 right: { left: x, opt: <=, right: 10 }
},
right: { left: x, opt: >, right: 20 }
}

Una única **semántica**

Dado un x, queremos aquellos valores de x donde la condición $(x > 0 \wedge x \leq 10) \vee (x > 20)$ se cumpla



Nuestro objetivo - **¿Qué buscamos?**

Tres puntos claves para construir
ese “algo”

- 01** Única
Expresión canónica
que represente una condición
booleana
- 02** **Mecanismo de**
representación para
cualquiera de sus variaciones
- 03** **Sintaxis friendly e**
idiomática

Preparando ese “algo” -
¿Cómo lo hacemos?

Un **DSL**
en apuros

DSL

Receta

Conceptos previos

Primer ingrediente - Gramática

- Conjunto de palabras / expresiones únicas
- Reglas para componer esas expresiones
- Cerrado, es decir, no admite expresiones fuera de la gramática

Ejemplos:

Sea **Bar** la gramática de las acciones de **Beber** en un bar

```
Bar    := Beber <Bebida> | Y <Bar> <Bar>
Bebida := Agua  | Zumo | Refresco | Cocktail ...
```

=>

```
expr1 := Beber Agua
expr2 := Beber Zumo
expr3 := Y (Beber Cocktail) (Beber Refresco)
```



DSL

Receta

Conceptos previos

Segundo ingrediente - Semántica

- Interpretaciones de las expresiones de la gramática
- Diferente semántica, diferente interpretación
- Aporta significado a nuestras expresiones

Ejemplos:

En nuestro bar están nuestros amigos *Andrea, David y Miguel*, cada uno quiere una cosa diferente...

A (Semántica Andrea)
D (Semántica David)
M (Semántica Miguel)

A(Beber Refresco) => beber A(Refresco) => beber Coca Cola
A(Beber Cocktail) => beber A(Cocktail) => beber Bloody Mary

D(Beber Refresco) => beber D(Refresco) => beber Fanta naranja
D(Beber Cocktail) => beber D(Cocktail) => beber Margarita

M(Beber Refresco) => beber M(Refresco) => beber Tónica
M(Beber Cocktail) => beber M(Cocktail) => beber Cosmopolitan





Criteria4s - Gramática + Semántica

Nuestra mezcla perfecta

- Gramática de expresiones booleanas y predicados
- **Unicidad** en las expresiones (*expresión canónica*)
- Diferente semántica, diferente interpretación, es decir, diferente **representación**

```
Criteria    ::=  <Conjunction> <Criteria> <Criteria> | <Predicate> <Ref> <Ref> | Value<Boolean>
Conjunction ::=  AND | OR
Predicate   ::=  EQ | NEQ | GT | LT | GEQ | LEQ | IN | LIKE | IS_NULL | IS_NOT_NULL ...
Ref         ::=  Value<T> | Col
```

```
OR (AND (GT x 0) (LEQ x 10)) (GT x 20)
// 1: - (('x' > 0) AND ('x' <= 10)) OR ('x' > 20)
// 2: - { ope: `OR`, left: { ope: `AND`, left: { ope: `gt` ,
//           left: `x`, right: '0' }, right: { ope: `leq`, left: `x` ,
//           right: '10' } }, right: { ope: `gt`, left: `x` , right: '20' } }
```

Criteria4s - Primer intento

PLANTEAMIENTO

- Usando tipos String
- Usando trait
- Usando singleton

```
trait Criteria {  
    def and(x: String, y: String): String  
    def or(x: String, y: String): String  
    def leq(x: String, y: String): String  
    def gt(x: String, y: String): String  
}
```

```
object Criteria {  
    def and(x: String, y: String): String = s"($x AND $y)"  
    def or(x: String, y: String): String = s"($x OR $y)"  
    def leq(x: String, y: String): String = s"$x <= $y"  
    def gt(x: String, y: String): String = s"$x > $y"  
}
```



Criteria4s - Primer intento

PLANTEAMIENTO

- Usando tipos String
- Usando trait
- Usando singleton

```
trait Criteria {  
    def and(x: String, y: String): String  
    def or(x: String, y: String): String  
    def leq(x: String, y: String): String  
    def gt(x: String, y: String): String  
}
```

```
object Criteria {  
    def and(x: String, y: String): String = s"($x AND $y)"  
    def or(x: String, y: String): String = s"($x OR $y)"  
    def leq(x: String, y: String): String = s"$x <= $y"  
    def gt(x: String, y: String): String = s"$x > $y"  
}
```



RESULTADOS

- ✖ Múltiple intérpretes

```
object DB1 extends Criteria {  
    def and(x: String, y: String): String = s"($x AND $y)"  
    def or(x: String, y: String): String = s"($x OR $y)"  
    def leq(x: String, y: String): String = s"$x <= $y"  
    def gt(x: String, y: String): String = s"$x > $y"  
}  
  
object DB2 extends Criteria {  
    def and(x: String, y: String): String = s"{$ope: `AND`, left: $x, right: $y}"  
    def or(x: String, y: String): String = s"{$ope: `OR`, left: $x, right: $y}"  
    def leq(x: String, y: String): String = s"{$ope: `leq`, left: $x, right: $y}"  
    def gt(x: String, y: String): String = s"{$ope: `gt`, left: $x, right: $y}"  
}
```

Criteria4s - Primer intento

PLANTEAMIENTO

- Usando tipos String
- Usando trait
- Usando singleton

```
trait Criteria {  
    def and(x: String, y: String): String  
    def or(x: String, y: String): String  
    def leq(x: String, y: String): String  
    def gt(x: String, y: String): String  
}
```

```
object Criteria {  
    def and(x: String, y: String): String = s"($x AND $y)"  
    def or(x: String, y: String): String = s"($x OR $y)"  
    def leq(x: String, y: String): String = s"$x <= $y"  
    def gt(x: String, y: String): String = s"$x > $y"  
}
```



RESULTADOS

- ✖ Múltiple intérpretes
- ✖ Cualquier String en sus parámetros
- ✖ Expresiones inválidas
- ✖ Error runtime!

```
import Interpreters.DB1._  
  
val expr1: String =  
    or(and(gt("x", "0"), leq("x", "10")), gt("x", "20")) // (x > 0 AND x <= 10) OR x > 20  
  
val expr2: String =  
    or(  
        and(  
            gt("x > 0", "x <= 10"),  
            leq(  
                "Lorem ipsum dolor sit amet ... ",  
                "En un lugar de la Mancha ... "  
            )  
        ),  
        "x + & * //"  
    ) // ((x > 0 > x <= 10 AND Lorem ipsum dolor sit amet ... <= En un lugar de la Mancha ... ) OR x + & * //")
```

Criteria4s - Primer intento

PLANTEAMIENTO

- Usando tipos String
- Usando trait
- Usando singleton

```
trait Criteria {  
    def and(x: String, y: String): String  
    def or(x: String, y: String): String  
    def leq(x: String, y: String): String  
    def gt(x: String, y: String): String  
}
```

```
object Criteria {  
    def and(x: String, y: String): String = s"($x AND $y)"  
    def or(x: String, y: String): String = s"($x OR $y)"  
    def leq(x: String, y: String): String = s"$x <= $y"  
    def gt(x: String, y: String): String = s"$x > $y"  
}
```



RESULTADOS

- ✖ Múltiple intérpretes
- ✖ Cualquier String en sus parámetros
- ✖ Expresiones inválidas
- ✖ Error runtime!
- ✖ Convivencia
- Gramática y semántica
- DB1.or /= DB2.or

```
import Interpreters._  
  
val exprDB1: String =  
    DB1.or(DB1.and(DB1.gt("x", "0"), DB1.leq("x", "10")), DB1.gt("x", "20"))  
  
val exprDB2: String =  
    DB2.or(DB2.and(DB2.gt("x", "0"), DB2.leq("x", "10")), DB2.gt("x", "20"))  
  
val expr1: String =  
    or(and(gt("x", "0"), leq("x", "10")), gt("x", "20")) // (x > 0 AND x <= 10) OR x > 20  
    ^~~~~~  
    Cannot resolve overruled method 'or'  
val expr2:  
    or  
    <: 1/2 >  
    <: io.github.ratfdr.criteria4s.presentation.FirstApproach.Interpreters.DB1 >  
    def or(x: String, y: String): String  
        ^~~~~~  
        criteria4s-presentation  
        ^~~~~~  
        Ø :  
    leg(  
        "Lorem ipsum dolor sit amet ... ",  
        "En un lugar de la Mancha ... "  
    )  
),  
    "x + 6 * /*  
) // ((x > 0 AND x <= 10 AND Lorem ipsum dolor sit amet ... & En un lugar de la Mancha ...) OR x + 6 * //)
```

Criteria4s - Segundo intento

PLANTEAMIENTO

→ Usando ADTs

```
sealed trait Criteria
sealed trait Conjunction extends Criteria
sealed trait Predicate extends Criteria

sealed trait Reference
case class Column(name: String) extends Reference
sealed trait Value[T]
case class Literal[T](value: T) extends Value[T]
case object True extends Reference with Value[Boolean]
case object False extends Reference with Value[Boolean]
case class And(left: Criteria, right: Criteria) extends Conjunction
case class Or(left: Criteria, right: Criteria) extends Conjunction
case class Leq(left: Reference, right: Reference) extends Predicate
case class Gt(left: Reference, right: Reference) extends Predicate
case class Eq(left: Reference, right: Reference) extends Predicate
case class IsNull(ref: Reference) extends Predicate
```



Criteria4s - Segundo intento

PLANTEAMIENTO

→ Usando ADTs

```
sealed trait Criteria
sealed trait Conjunction extends Criteria
sealed trait Predicate extends Criteria

sealed trait Reference
case class Column(name: String) extends Reference
sealed trait Value[T] extends Reference
case class Literal[T](value: T) extends Value[T]
case object True extends Reference with Value[Boolean]
case object False extends Reference with Value[Boolean]
case class And(left: Criteria, right: Criteria) extends Conjunction
case class Or(left: Criteria, right: Criteria) extends Conjunction
case class Leq(left: Reference, right: Reference) extends Predicate
case class Gt(left: Reference, right: Reference) extends Predicate
case class Eq(left: Reference, right: Reference) extends Predicate
case class IsNull(ref: Reference) extends Predicate
```



RESULTADOS

✓ Múltiple intérpretes

```
/** DB1 */
object DB1 {
  private def evalR(r: Reference): String = r match {
    case Column(name) => s"$name"
    case Literal(v)   => s"$v"
    case True         => "true"
    case False        => "false"
  }
  def eval(c: Criteria): String = c match {
    case And(l, r) => s"(${eval(l)} AND ${eval(r)})"
    case Or(l, r)  => s"(${eval(l)} OR ${eval(r)})"
    case Leq(l, r) => s"${evalR(l)} <= ${evalR(r)}"
    case Gt(l, r)  => s"${evalR(l)} > ${evalR(r)}"
    case Eq(l, r)  => s"${evalR(l)} = ${evalR(r)}"
    case IsNull(r) => s"${evalR(r)} IS NULL"
  }
}
```

```
/** DB2 */
object DB2 {
  private def evalR(r: Reference): String = r match {
    case Column(name) => s"$name"
    case Literal(v)   => s"$v"
    case True         => "true"
    case False        => "false"
  }
  def eval(c: Criteria): String = c match {
    case And(l, r) => s"${ope: 'AND', left: ${eval(l)}, right: ${eval(r)}}"
    case Or(l, r)  => s"${ope: 'OR', left: ${eval(l)}, right: ${eval(r)}}"
    case Leq(l, r) => s"${ope: 'leq', left: ${evalR(l)}, right: ${evalR(r)}}"
    case Gt(l, r)  => s"${ope: 'gt', left: ${evalR(l)}, right: ${evalR(r)}}"
    case Eq(l, r)  => s"${ope: 'eq', left: ${evalR(l)}, right: ${evalR(r)}}"
  }
}
```

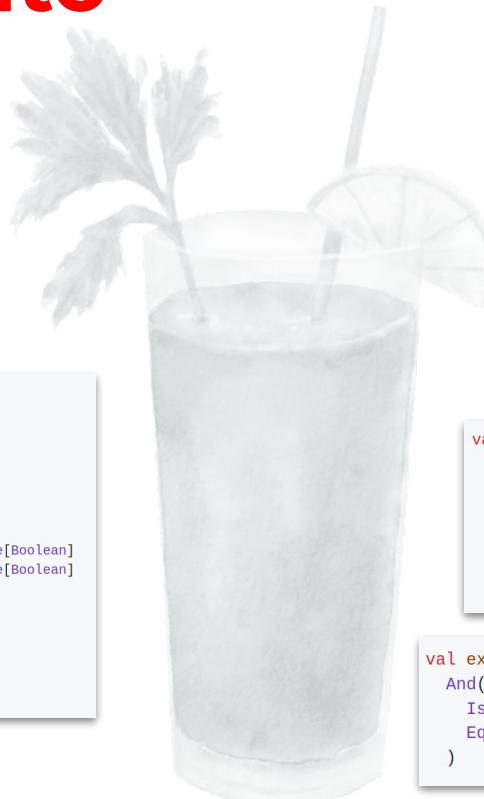
Criteria4s - Segundo intento

PLANTEAMIENTO

→ Usando ADTs

```
sealed trait Criteria
sealed trait Conjunction extends Criteria
sealed trait Predicate extends Criteria

sealed trait Reference
case class Column(name: String) extends Reference
sealed trait Value[T]
case class Literal[T](value: T) extends Value[T]
case object True extends Reference with Value[Boolean]
case object False extends Reference with Value[Boolean]
case class And(left: Criteria, right: Criteria) extends Conjunction
case class Or(left: Criteria, right: Criteria) extends Conjunction
case class Leq(left: Reference, right: Reference) extends Predicate
case class Gt(left: Reference, right: Reference) extends Predicate
case class Eq(left: Reference, right: Reference) extends Predicate
case class IsNull(ref: Reference) extends Predicate
```



RESULTADOS

- ✓ Múltiple intérpretes
- ✓ Sólo se puede usar ADTs
- ✓ Expresiones válidas
- Gramática cerrada

```
val expr: Criteria =
  Or(
    And(
      Gt(Column("x"), Literal(0)),
      Leq(Column("x"), Literal(10))
    ),
    Gt(Column("x"), Literal(20))
  )
```

```
val expr2: Criteria =
  Or(
    And(
      Gt(Column("x"), Literal(0)),
      Leq(Column("x"), Literal(10))
    ),
    Eq(Column("user"), Literal("07715cee-5d87-427d-99a7-cc03f2b5ef4a"))
  )
```

```
val expr3: Criteria =
  And(
    IsNull(Column("user")),
    Eq(Column("user"), Literal("07715cee-5d87-427d-99a7-cc03f2b5ef4a"))
  )
```

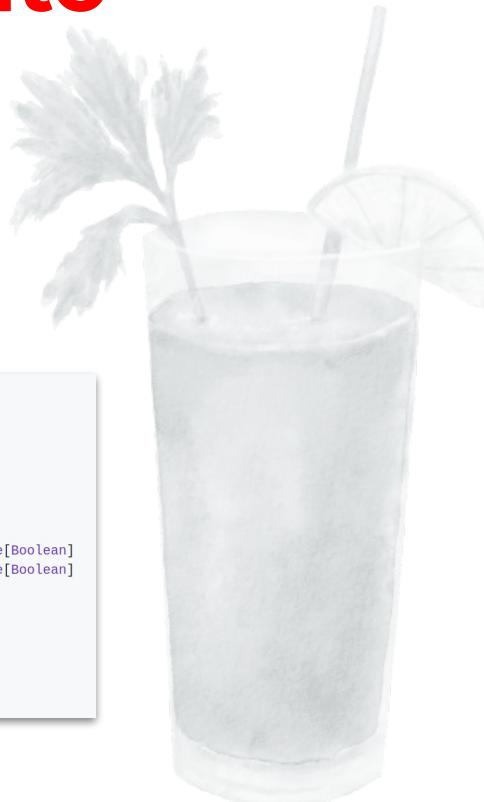
Criteria4s - Segundo intento

PLANTEAMIENTO

→ Usando ADTs

```
sealed trait Criteria
sealed trait Conjunction extends Criteria
sealed trait Predicate extends Criteria

sealed trait Reference
case class Column(name: String) extends Reference
sealed trait Value[T]
case class Literal[T](value: T) extends Reference with Value[T]
case object True extends Reference with Value[Boolean]
case object False extends Reference with Value[Boolean]
case class And(left: Criteria, right: Criteria) extends Conjunction
case class Or(left: Criteria, right: Criteria) extends Conjunction
case class Leq(left: Reference, right: Reference) extends Predicate
case class Gt(left: Reference, right: Reference) extends Predicate
case class Eq(left: Reference, right: Reference) extends Predicate
case class IsNull(ref: Reference) extends Predicate
```



RESULTADOS

- ✖ Expression problem
- Añadir nueva expresión
- Between
- Isn
- ...

```
case class Eq(left: Reference, right: Reference) extends Predicate
case class IsNull(ref: Reference) extends Predicate
/* ... */
case class IsIn(left: Reference, right: Reference) extends Predicate
case class Between(left: Reference, right: Reference) extends Predicate
```

```
object DB1 {
  def eval(c: Criteria): String = c match {
    /* ... */
    case Eq(l, r) => s"${evalR(l)} = ${evalR(r)}"
    /* ... */
    case IsIn(l, r) => s"${evalR(l)} isin ${evalR(r)}"
    case Between(l, r) => s"${evalR(l)} between ${evalR(r)}"
  }
}
```

```
object DB2 {
  def eval(c: Criteria): String = c match {
    /* ... */
    case Eq(l, r) => s"[ ope: 'eq', left: ${evalR(l)}, right: ${evalR(r)} ]"
    /* ... */
    case IsIn(l, r) => s"[ ope: 'isin', left: ${evalR(l)}, right: ${evalR(r)} ]"
    case Between(l, r) => s"[ ope: 'between', left: ${evalR(l)}, right: ${evalR(r)} ]"
  }
}
```

```
object DB3 {
  def eval(c: Criteria): String = c match {
    /* ... */
    case Eq(l, r) => /* implemented */
    /* ... */
    case IsIn(l, r) => /* implemented */
    case Between(l, r) => /* implemented */
  }
}
```

Criteria4s - Segundo intento

PLANTEAMIENTO

→ Usando ADTs

```
sealed trait Criteria
sealed trait Conjunction extends Criteria
sealed trait Predicate extends Criteria

sealed trait Reference
case class Column(name: String) extends Reference
sealed trait Value[T]
case class Literal[T](value: T) extends Value[T]
case object True extends Reference with Value[Boolean]
case object False extends Reference with Value[Boolean]
case class And(left: Criteria, right: Criteria) extends Conjunction
case class Or(left: Criteria, right: Criteria) extends Conjunction
case class Leq(left: Reference, right: Reference) extends Predicate
case class Gt(left: Reference, right: Reference) extends Predicate
case class Eq(left: Reference, right: Reference) extends Predicate
case class IsNull(ref: Reference) extends Predicate
```



RESULTADOS

- ✖ Expression problem
Añadir nueva expresión
→ Between
→ Isn't
→ ...

- ✖ Expresión que no se usa en un datastore
Error en runtime!

```
val expr3: Criteria =
  And(
    IsNull(Column("user")),
    Eq(Column("user"), Literal("07715cee-5d87-427d-99a7-cc03f2b5ef4a"))
  )
```

```
/** DB2.2 */
object DB22 {
  /* ... */
  def eval(c: Criteria): String = c match {
    case And(l, r) => s"{ ope: 'AND', left: ${eval(l)}, right: ${eval(r)} }"
    case Or(l, r) => s"{ ope: 'OR', left: ${eval(l)}, right: ${eval(r)} }"
    case Leq(l, r) => s"{ ope: 'leq', left: ${evalR(l)}, right: ${evalR(r)} }"
    case Gt(l, r) => s"{ ope: 'gt', left: ${evalR(l)}, right: ${evalR(r)} }"
    case Eq(l, r) => s"{ ope: 'eq', left: ${evalR(l)}, right: ${evalR(r)} }"
    case _ => ??? // Exception? None? Either?
  }
}
```

```
/** Expr 3 */
DB2.eval(expr3)
// ('user' IS NULL AND `user` = '07715cee-5d87-427d-99a7-cc03f2b5ef4a')

DB2.eval(expr3)
// This will fail because IsNull is not implemented in DB2 interpreter!
```

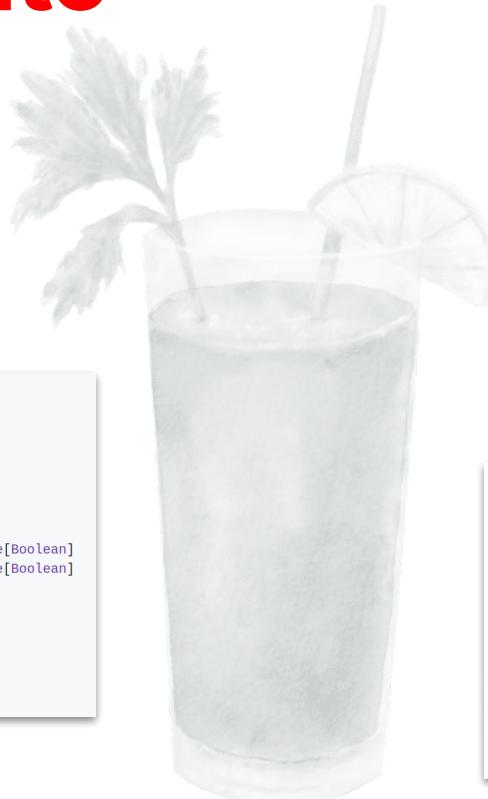
Criteria4s - Segundo intento

PLANTEAMIENTO

→ Usando ADTs

```
sealed trait Criteria
sealed trait Conjunction extends Criteria
sealed trait Predicate extends Criteria

sealed trait Reference
case class Column(name: String) extends Reference
sealed trait Value[T]
case class Literal[T](value: T) extends Value[T]
case object True extends Reference with Value[Boolean]
case object False extends Reference with Value[Boolean]
case class And(left: Criteria, right: Criteria) extends Conjunction
case class Or(left: Criteria, right: Criteria) extends Conjunction
case class Leq(left: Reference, right: Reference) extends Predicate
case class Gt(left: Reference, right: Reference) extends Predicate
case class Eq(left: Reference, right: Reference) extends Predicate
case class IsNull(ref: Reference) extends Predicate
```



RESULTADOS

✖ Expression problem
Añadir nueva expresión
→ Between
→ Isn't
→ ...

✖ Expresión que no se usa en un datastore
Error en runtime!

✖ Convivencia
→ Gramática
→ Semántica

```
import DB1._
import DB2._

val expr: Criteria =
  or(
    And(
      Gt(Column("x"), Literal(0)),
      Leq(Column("x"), Literal(10))
    ),
    Gt(Column("x"), Literal(20))
  )

eval(expr)
```

```
val expr: Criteria =
  or(
    And(
      Gt(Column("x"), Literal(0)),
      Leq(Column("x"), Literal(10))
    ),
    Gt(Column("x"), Literal(20))
  )

eval(expr)
```

```
DB1.eval(expr)

DB2.eval(expr)
```

El ingrediente secreto -
La solución

**TYPE
CLASSES**
al rescate

Criteria4s - Intento final - Type classes

PLANTEAMIENTO

```
trait Criteria[D] {  
    def value: String  
}  
  
trait Conjunction[D] {  
    def eval(cr1: Criteria[D], cr2: Criteria[D]): Criteria[D]  
}  
  
trait OR[D] extends Conjunction[D]  
trait AND[D] extends Conjunction[D]  
  
trait Predicate[D]  
trait PredicateBinary[D] extends Predicate[D] {  
    def eval(cr1: Reference[D], cr2: Reference[D]): Criteria[D]  
}  
  
trait PredicateUnary[D] extends Predicate[D] {  
    def eval(cr: Reference[D]): Criteria[D]  
}  
  
trait Gt[D] extends PredicateBinary[D]  
trait Leq[D] extends PredicateBinary[D]  
trait Eq[D] extends PredicateBinary[D]  
trait IsNull[D] extends PredicateUnary[D]  
  
trait Reference[D] {  
    def value: String  
}  
  
trait Column[D] extends Reference[D]  
trait Value[D, V] extends Reference[D]
```



Criteria4s - Intento final: Type classes

PLANTEAMIENTO

```
trait Criteria[D] {
    def value: String
}

trait Conjunction[D] {
    def eval(cr1: Criteria[D], cr2: Criteria[D]): Criteria[D]
}

trait OR[D] extends Conjunction[D]
trait AND[D] extends Conjunction[D]

trait Predicate[D]
trait PredicateBinary[D] extends Predicate[D] {
    def eval(cr1: Reference[D], cr2: Reference[D]): Criteria[D]
}

trait PredicateUnary[D] extends Predicate[D] {
    def eval(cr: Reference[D]): Criteria[D]
}

trait Gt[D] extends PredicateBinary[D]
trait Leq[D] extends PredicateBinary[D]
trait Eq[D] extends PredicateBinary[D]
trait IsNull[D] extends PredicateUnary[D]

trait Reference[D] {
    def value: String
}

trait Column[D] extends Reference[D]
trait Value[D, V] extends Reference[D]
```



RESULTADOS

- ✓ Múltiple intérpretes

```
trait DB1 // DB1 type

object DB1 {
    implicit val orDB1: OR[DB1] = new OR[DB1] {
        override def eval(cr1: Criteria[DB1], cr2: Criteria[DB1]): Criteria[DB1] =
            Criteria.pure(s"(${cr1.value} OR ${cr2.value})")
    }
    implicit val andDB1: AND[DB1] = new AND[DB1] {
        override def eval(cr1: Criteria[DB1], cr2: Criteria[DB1]): Criteria[DB1] =
            Criteria.pure(s"(${cr1.value} AND ${cr2.value})")
    }
    implicit val greaterDB1: Gt[DB1] = new Gt[DB1] {
        override def eval(cr1: Reference[DB1], cr2: Reference[DB1]): Criteria[DB1] =
            Criteria.pure(s"${cr1.value} > ${cr2.value}")
    }
    implicit val lessDB1: Leq[DB1] = new Leq[DB1] {
        override def eval(cr1: Reference[DB1], cr2: Reference[DB1]): Criteria[DB1] =
            Criteria.pure(s"${cr1.value} <= ${cr2.value}")
    }
    implicit val equalDB1: Eq[DB1] = new Eq[DB1] {
        override def eval(cr1: Reference[DB1], cr2: Reference[DB1]): Criteria[DB1] =
            Criteria.pure(s"${cr1.value} = ${cr2.value}")
    }
    implicit val isNullDB1: IsNull[DB1] = new IsNull[DB1] {
        override def eval(cr: Reference[DB1]): Criteria[DB1] = Criteria.pure(s"${cr.value} IS NULL")
    }
}
```

```
trait DB2 // DB2 type

object DB2 {
    implicit val orDB2: OR[DB2] = new OR[DB2] {
        override def eval(cr1: Criteria[DB2], cr2: Criteria[DB2]): Criteria[DB2] =
            Criteria.pure(s"(${cr1.value} OR ${cr2.value})")
    }
    implicit val andDB2: AND[DB2] = new AND[DB2] {
        override def eval(cr1: Criteria[DB2], cr2: Criteria[DB2]): Criteria[DB2] =
            Criteria.pure(s"(${cr1.value} AND ${cr2.value})")
    }
    implicit val gtDB2: Gt[DB2] = new Gt[DB2] {
        override def eval(cr1: Reference[DB2], cr2: Reference[DB2]): Criteria[DB2] =
            Criteria.pure(s"${cr1.value} > ${cr2.value}")
    }
    implicit val leqDB2: Leq[DB2] = new Leq[DB2] {
        override def eval(cr1: Reference[DB2], cr2: Reference[DB2]): Criteria[DB2] =
            Criteria.pure(s"${cr1.value} <= ${cr2.value}")
    }
    implicit val eqDB2: Eq[DB2] = new Eq[DB2] {
        override def eval(cr1: Reference[DB2], cr2: Reference[DB2]): Criteria[DB2] =
            Criteria.pure(s"${cr1.value} = ${cr2.value}")
    }
}
```

Criteria4s - Intento final: Type classes

PLANTEAMIENTO

```
trait Criteria[D] {  
    def value: String  
}  
  
trait Conjunction[D] {  
    def eval(cr1: Criteria[D], cr2: Criteria[D]): Criteria[D]  
}  
  
trait OR[D] extends Conjunction[D]  
trait AND[D] extends Conjunction[D]  
  
trait Predicate[D]  
trait PredicateBinary[D] extends Predicate[D] {  
    def eval(cr1: Reference[D], cr2: Reference[D]): Criteria[D]  
}  
  
trait PredicateUnary[D] extends Predicate[D] {  
    def eval(cr: Reference[D]): Criteria[D]  
}  
  
trait Gt[D] extends PredicateBinary[D]  
trait Leq[D] extends PredicateBinary[D]  
trait Eq[D] extends PredicateBinary[D]  
trait IsNull[D] extends PredicateUnary[D]  
  
trait Reference[D] {  
    def value: String  
}  
  
trait Column[D] extends Reference[D]  
trait Value[D, V] extends Reference[D]
```



RESULTADOS

- ✓ Múltiple intérpretes
- ✓ Sólo se puede usar si es tipo de la Type Class
- ✓ Expresiones válidas
- ✓ Gramática cerrada

```
val expr2: Criteria =  
  Or(  
    And(  
      Gt(Column("x"), Literal(0)),  
      Leq(Column("x"), Literal(10))  
    ),  
    Eq(Column("user"), Literal("07715cee-5d87-427d-99a7-cc03f2b5ef4a"))  
  )
```

```
val expr3: Criteria =  
  And(  
    IsNull(Column("user")),  
    Eq(Column("user"), Literal("07715cee-5d87-427d-99a7-cc03f2b5ef4a"))  
  )
```

/* Examples */
def expr[D: Gt: Leq: AND: OR]: Criteria[D] =
 ((col[D]("x") >: lit(0)) and (col[D]("x") :<= lit(10))) or
 (col[D]("x") >: lit(20))

def exprIsNull[D: IsNull: OR: Eq]: Criteria[D] =
 col[D]("user").isNull or
 (col[D]("user") := lit("07715cee-5d87-427d-99a7-cc03f2b5ef4a"))

Criteria4s - Intento final: Type classes

PLANTEAMIENTO

```
trait Criteria[D] {  
    def value: String  
}  
  
trait Conjunction[D] {  
    def eval(cr1: Criteria[D], cr2: Criteria[D]): Criteria[D]  
}  
  
trait OR[D] extends Conjunction[D]  
trait AND[D] extends Conjunction[D]  
  
trait Predicate[D]  
trait PredicateBinary[D] extends Predicate[D] {  
    def eval(cr1: Reference[D], cr2: Reference[D]): Criteria[D]  
}  
  
trait PredicateUnary[D] extends Predicate[D] {  
    def eval(cr: Reference[D]): Criteria[D]  
}  
  
trait Gt[D] extends PredicateBinary[D]  
trait Leq[D] extends PredicateBinary[D]  
trait Eq[D] extends PredicateBinary[D]  
trait IsNull[D] extends PredicateUnary[D]  
  
trait Reference[D] {  
    def value: String  
}  
  
trait Column[D] extends Reference[D]  
trait Value[D, V] extends Reference[D]
```



RESULTADOS

- ✓ Múltiple intérpretes
- ✓ Sólo se puede usar ADTs
- ✓ Expresiones válidas
- Gramática cerrada
- No hay *expression problem*
- ✓ Expresión que no se usa en un datastore
- Error en compilación!**

```
trait DB1 // DB1 type  
object DB1 {  
    /* ... */  
    implicit val eqDB1: Eq[DB1] = new Eq[DB1] {  
        override def eval(cr1: Reference[DB1], cr2: Reference[DB1]): Criteria[DB1] =  
            Criteria.pure(s"${cr1.value} + ${cr2.value}")  
    }  
    implicit val isNullDB1: IsNull[DB1] = new IsNull[DB1] {  
        override def eval(cr: Reference[DB1]): Criteria[DB1] = Criteria.pure(s"${cr.value} IS NULL")  
    }  
}  
  
trait DB2 // DB2 type  
object DB2 {  
    /* ... */  
    implicit val eqDB2: Eq[DB2] = new Eq[DB2] {  
        override def eval(cr1: Reference[DB2], cr2: Reference[DB2]): Criteria[DB2] =  
            Criteria.pure(s"${cr1.value} * ${cr2.value}")  
    }  
}  
  
/* With IsNull expression */  
  
def exprIsNull[D: IsNull: OR: Eq]: Criteria[D] =  
    col[D]("user").isNull ||  
    (col[D]("user") := lit("07715cee-5d87-427d-99a7-cc03f2b5ef4a"))  
  
val exprIsNull1: Criteria[DB1] = exprIsNull[DB1]  
val exprIsNull2: Criteria[DB2] = exprIsNull[DB2](...)
```

Criteria4s - Intento final: Type classes

PLANTEAMIENTO

```
trait Criteria[D] {  
    def value: String  
}  
  
trait Conjunction[D] {  
    def eval(cr1: Criteria[D], cr2: Criteria[D]): Criteria[D]  
}  
  
trait OR[D] extends Conjunction[D]  
trait AND[D] extends Conjunction[D]  
  
trait Predicate[D]  
trait PredicateBinary[D] extends Predicate[D] {  
    def eval(cr1: Reference[D], cr2: Reference[D]): Criteria[D]  
}  
  
trait PredicateUnary[D] extends Predicate[D] {  
    def eval(cr: Reference[D]): Criteria[D]  
}  
  
trait Gt[D] extends PredicateBinary[D]  
trait Leq[D] extends PredicateBinary[D]  
trait Eq[D] extends PredicateBinary[D]  
trait IsNull[D] extends PredicateUnary[D]  
  
trait Reference[D] {  
    def value: String  
}  
  
trait Column[D] extends Reference[D]  
trait Value[D, V] extends Reference[D]
```



RESULTADOS

- ✓ Múltiple intérpretes
- ✓ Sólo se puede usar ADTs
- ✓ Expresiones válidas
- Gramática cerrada
- No hay *expression problem*
- ✓ Expresión que no se usa en un datastore
- Error en compilación!
- ✓ Convivencia

```
import Interpreters._  
  
/** Examples */  
def expr[D: Gt: Leq: AND: OR]: Criteria[D] =  
    ((col[D]("x") :> lit(0)) and (col[D]("x") :≤ lit(10))) or  
    (col[D]("x") :> lit(20))  
  
val expr1: Criteria[DB1] = expr[DB1]  
  
val expr2: Criteria[DB2] = expr[DB2]  
  
val exprInline: Criteria[DB1] =  
    ((col[DB1]("x") :> lit(0)) and (col[DB1]("x") :≤ lit(10))) or  
    (col[DB1]("x") :> lit(20))
```

Criteria4s - Intento final: Type classes + Tag

PLANTEAMIENTO

```
trait Criteria[D] {  
    def value: String  
}  
  
trait Conjunction[D] {  
    def eval(cr1: Criteria[D], cr2: Criteria[D]): Criteria[D]  
}  
  
trait OR[D] extends Conjunction[D]  
trait AND[D] extends Conjunction[D]  
  
trait Predicate[D]  
trait PredicateBinary[D] extends Predicate[D] {  
    def eval(cr1: Reference[D], cr2: Reference[D]): Criteria[D]  
}  
  
trait PredicateUnary[D] extends Predicate[D] {  
    def eval(cr: Reference[D]): Criteria[D]  
}  
  
trait Gt[D] extends PredicateBinary[D]  
trait Leq[D] extends PredicateBinary[D]  
trait Eq[D] extends PredicateBinary[D]  
trait IsNull[D] extends PredicateUnary[D]  
  
trait Reference[D] {  
    def value: String  
}  
  
trait Column[D] extends Reference[D]  
trait Value[D, V] extends Reference[D]
```



RESULTADOS

- Criteria[DB1]
- ✗ → Criteria[Boolean]
→ Criteria[Option[Int]]
→ Criteria[List[List[Int]]]
→ Criteria[Any]
→ ...
- ✓ Tag
 - DB1 <: Tag => Criteria[DB1]
 - DB2 <: Tag => Criteria[DB2]
 - ...

Criteria4s - Intento final: Type classes + Tag

PLANTEAMIENTO

```
trait Criteria[D] {  
    def value: String  
}  
  
trait Conjunction[D] {  
    def eval(crit1: Criteria[D], crit2: Criteria[D]): Criteria[D]  
}  
  
trait OR[D] extends Conjunction[D]  
trait AND[D] extends Conjunction[D]  
  
trait Predicate[D]  
  
trait PredicateBinary[D] extends Predicate[D] {  
    def eval(crit1: Reference[D], crit2: Reference[D]): Criteria[D]  
}  
  
trait PredicateUnary[D] extends Predicate[D] {  
    def eval(crit: Reference[D]): Criteria[D]  
}  
  
trait Gt[D] extends PredicateBinary[D]  
trait Leq[D] extends PredicateBinary[D]  
trait Eq[D] extends PredicateBinary[D]  
trait IsNull[D] extends PredicateUnary[D]  
  
trait Reference[D] {  
    def value: String  
}  
  
trait Column[D] extends Reference[D]  
trait Value[D, V] extends Reference[D]
```



RESULTADOS (ilustrativos)



```
/** These expressions are too confusing */  
val exprString: Criteria[String] =  
    ((col[String]("x") :> lit(0)) and (col[String]("x") :≤ lit(10))) or  
    (col[String]("x") :> lit(20))  
  
val exprOptionInt: Criteria[Option[Int]] =  
    ((col[Option[Int]]("x") :> lit(0)) and (col[Option[Int]]("x") :≤ lit(10))) or  
    (col[Option[Int]]("x") :> lit(20))
```



```
/** Examples */  
def expr[D <: CriteriaTag: Gt: Leq: AND: OR]: Criteria[D] =  
    ((col[D]("x") :> lit(0)) and (col[D]("x") :≤ lit(10))) or  
    (col[D]("x") :> lit(20))  
  
def exprIsNull[D <: CriteriaTag: IsNull: OR: Eq]: Criteria[D] =  
    col[D]("user").isNull or  
    (col[D]("user") := lit("07715cee-5d87-427d-99a7-cc03f2b5ef4a"))
```

La bebida perfecta -
El resultado

**MOMENTO
POC**

Criteria4s - Resultados

¿Cómo funciona?

Requisitos

- Expresión del DSL
- Intérprete de la DB

```
(col[MongoDB]("owner") === lit("owner")) and (col[MongoDB]("rate") gt lit(4.5))

(col[Postgres]("owner") === lit("owner")) and (col[Postgres]("rate") gt lit(4.5))
```

```
def criteria[D <: CriteriaTag: EQ: GT: Sym: AND](owner: String, rate: Double): Criteria[D] =  
  (col[D]("owner") === lit(owner)) and (col[D]("rate") gt lit(rate))

criteria[MongoDB]("owner", 4.5)

criteria[Postgres]("owner", 4.5)
```

RESULTADOS

- Criterio

```
{ $and: [ { owner: { $eq: "owner" } }, { rate: { $gt: 4.5 } } ] }
```

```
(owner = 'owner') AND (rate > '4.5')
```



Criteria4s - Resultados

¿Cómo funciona?

Requisitos

→ Implementar el intérprete de la DB

```
trait WeirdDatastore extends CriteriaTag

object WeirdDatastore {

  private def wone(symbol: String)(left: String, right: String): String =
    s"""$left $symbol $right" stripMargin

  private def wope(symbol: String)(left: String): String =
    s"""$left $symbol $right" stripMargin

  implicit def showSeq[V](implicit show: Show[V, WeirdDatastore]): Show[Seq[V], WeirdDatastore] = Show.create(_ map (show.show).mkString("[", " ", "]"))

  implicit def showRange[V](implicit show: Show[V, WeirdDatastore]): Show[Seq[V], WeirdDatastore] = Show.create(_ case (l, r) => s"${show.show(l)}, ${show.show(r)}" *)

  implicit val showColumn: Show[Column, WeirdDatastore] = Show.create(_ colName)

  implicit val ltPred: LT[WeirdDatastore] = build[WeirdDatastore, LT](wope("<"))

  implicit val gtPred: GT[WeirdDatastore] = build[WeirdDatastore, GT](wope(">"))

  implicit val orOp: OR[WeirdDatastore] = build[WeirdDatastore, OR](wope("OR"))

}

trait MongoDB extends CriteriaTag

object MongoDB {

  val s: String      = "$"
  val prefix: String = "$C#"

  private def dropPrefix(s: String): String = s.replace(prefix, "")

  private def format(symbol: String)(left: String, right: String): String = {
    if (left.startsWith(prefix))
      s"""$dropPrefix(left) ${innerF(symbol)(right)} $right" stripMargin
    else if (right.startsWith(prefix))
      s"""$dropPrefix(right) ${innerF(symbol)(left)} $left" stripMargin
    else s"""( $symbol: $left $right )" stripMargin
  }

  private def innerF(symbol: String)(left: String): String =
    s"""( $symbol: $left )" stripMargin

  implicit val ltPred: LT[MongoDB] = build[MongoDB, LT](format("lt"))

  implicit val gtPred: GT[MongoDB] = build[MongoDB, GT](format("gt"))

  implicit val andOp: AND[MongoDB] = build[MongoDB, AND](format("and"))

  implicit val eqPred: EQ[MongoDB] = build[MongoDB, EQ](format("eq"))

  implicit val neqPred: NEQ[MongoDB] = build[MongoDB, NEQ](format("ne"))

  implicit val leqPred: LEQ[MongoDB] = build[MongoDB, LEQ](format("lte"))

  implicit val geqPred: GEQ[MongoDB] = build[MongoDB, GEQ](format("gte"))

}
```

```
trait Postgres extends SQL
object Postgres extends SQLExpr[Postgres] {

  /**
   * That's not the right symbol for Postgres but it's just an example of how
   * default implementation
   */

  implicit def showSeq[T](implicit show: Show[T, Postgres]): Show[Seq[T], Postgres] = Show.create(_ map (show.show).mkString(", ", ", "))

  implicit def betweenShow[T](implicit show: Show[T, Postgres]): Show[(T, T), Postgres] = Show.create { case (l, r) => s"${show.show(l)} TO ${show.show(r)}" }

}
```

RESULTADOS

→ Criterio

```
(col[MongoDB]("owner") === lit("owner")) and (col[MongoDB]("rate") gt lit(4.5))

(col[Postgres]("owner") === lit("owner")) and (col[Postgres]("rate") gt lit(4.5))
```

```
def criteria[D <: CriteriaTag: EQ: GT: Sym: AND](owner: String, rate: Double): Criteria[D] =
  (col[D]("owner") === lit(owner)) and (col[D]("rate") gt lit(rate))

criteria[MongoDB]("owner", 4.5)

criteria[Postgres]("owner", 4.5)
```

```
{ $and: [ { owner: { $eq: "owner" } }, { rate: { $gt: 4.5 } } ] }
```

```
(owner = 'owner') AND (rate > '4.5')
```

Criteria4s - Resultados

Respondiendo a CAPA 2 = Repository (III)

PROBLEMA

- Generalizar / unificar el criterio
- ¿Cómo se va a expresar?
- ¿Qué forma va a tener? (*String, ADT, etc.*)

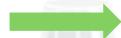
```
trait ReviewRepository[F[_]] {  
  
  def getReviews: F[Seq[Review]]  
  
  def getReviewBy(criteria: String): F[Seq[Review]]  
  
}
```

```
trait ReviewRepository[F[_]] {  
  
  def getReviews: F[Seq[Review]]  
  
  def getReviewBy(criteria: Criteria): F[Seq[Review]]  
  
}
```

RESULTADOS

- Criteria4s
- DSL que generaliza criterios
- Type classes

```
trait ReviewRepository[F[_]] {  
  
  def getReviews: F[Seq[Review]]  
  
  def getReviewBy[D <: CriteriaTag](criteria: Criteria[D]): F[Seq[Review]]  
  
}
```



Criteria4s - Resultados

Respondiendo a CAPA 2 = Repository (II)

PROBLEMA

- Aumentan los criterios
- Diferentes DB
- Diferentes expresiones de criterios

```
override def getReviewBy(rate: Double): F[Seq[Review]] =  
  execute(jdbc, s"SELECT * FROM $Airbnb.$Reviews WHERE rate > '$rate'") { rs =>  
    resultSetToReview(rs)  
  }  
  
override def getReviewBy(owner: String, rate: Double): F[Seq[Review]] =  
  execute(jdbc, s"SELECT * FROM $Airbnb.$Reviews WHERE (owner = '$owner') AND (rate > '$rate')") { rs =>  
    resultSetToReview(rs)  
  }  
  
override def getReviewBy(rate: Double): F[Seq[Review]] =  
  for {  
    docs ← Async[F].fromFuture(  
      collection.find(Document(s"**{ rate: { $gt: $rate } **")).toFuture().pure[F])  
    reviews = docs.map(documentToReview)  
  } yield reviews  
  
override def getReviewBy(owner: String, rate: Double): F[Seq[Review]] =  
  for {  
    docs ← Async[F].fromFuture(  
      collection.find(Document(s"**{ $and: [ { owner: { $eq: \"$owner\" } }, { rate: { $gt: $rate } } ] } **").  
        toFuture().pure[F])  
    reviews = docs.map(documentToReview)  
  } yield reviews
```

RESULTADOS

- Agnóstico a la DB
- Parametrizado por un criterio

```
/* MongoDB ReviewRepository implementation */  
def using[F[_]: Async: Logger](mongo: MongoClient): ReviewRepository[F] =  
  new ReviewRepository[F] {  
  
    def db: MongoDB                         = mongo.getDatabase(Airbnb)  
    def collection: MongoCollection[Document] = db.getCollection(Reviews)  
  
    override def getReviews: F[Seq[Review]] =  
      for {  
        docs ← Async[F].fromFuture(collection.find().toFuture().pure[F])  
        reviews = docs.map(documentToReview)  
      } yield reviews  
  
    override def getReviewByID(criterias: Criteria[D]): F[Seq[Review]] =  
      for {  
        _ ← logger[F].info(s"Searching $Reviews by criteria: ${criterias}")  
        docs ← Async[F].fromFuture(  
          collection.find(Document(criterias.toString())).toFuture().pure[F]  
        )  
        reviews = docs.map(documentToReview)  
      } yield reviews  
  
    /* JDBC ReviewRepository implementation */  
    def using[F[_]: Async: Logger](jdbc: Connection): ReviewRepository[F] =  
      new ReviewRepository[F] {  
  
        override def getReviews: F[Seq[Review]] =  
          execute(jdbc, s"SELECT * FROM $Airbnb.$Reviews") { rs =>  
            resultSetToReview(rs)  
          }  
  
        override def getReviewBy(D: CriteriaTag)(criterias: Criteria[D]): F[Seq[Review]] =  
          logger[F].info(s"Searching $Reviews by criteria: ${criterias}") =>  
          execute(jdbc, s"SELECT * FROM $Airbnb.$Reviews WHERE ${criterias}") { rs =>  
            resultSetToReview(rs)  
          }  
      }
```

Criteria4s - Resultados

Respondiendo a CAPA 2 = Repository (II)

PROBLEMA

→ No escala

```
trait ReviewRepository[F[_]] {  
  def getReviews: F[Seq[Review]]  
  
  def getReviewByOwner(owner: String): F[Seq[Review]]  
  
  def getReviewByRate(rate: Double): F[Seq[Review]]  
  
  def getReviewByOwnerAndRate(owner: String, rate: Double): F[Seq[Review]]  
  
  def getReviewByAddress(address: String): F[Seq[Review]]  
  
  def getReviewByRateAndAddress(rate: Double, address: String): F[Seq[Review]]  
  
  def getReviewByOwnerAndAddress(owner: String, address: String): F[Seq[Review]]  
  
  def getReviewByOwnerRateAndAddress(owner: String, rate: Double, address: String): F[Seq[Review]]  
  
  def getReviewByComment(comment: String): F[Seq[Review]]  
  
  def getReviewByRateAndComment(rate: Double, comment: String): F[Seq[Review]]  
  
  def getReviewByOwnerAndComment(owner: String, comment: String): F[Seq[Review]]  
  
  def getReviewByOwnerRateAndComment(owner: String, rate: Double, comment: String): F[Seq[Review]]  
  
  def getReviewByAddressAndComment(address: String, comment: String): F[Seq[Review]]  
}
```



```
/** Generic ReviewServices implementation */  
def build[F[_]: Async: Logger, D <: CriteriaTag: EQ: GT: Sym: AND](  
  rep: ReviewRepository[F]  
): ReviewServices[F] =  
  new ReviewServices[F] {  
  
  override def getAllReviews: F[Seq[Review]] =  
    Logger[F].info("Getting all reviews") => rep.getReviews  
  
  override def getReviewBy(owner: String): F[Seq[Review]] =  
    Logger[F].info("Getting reviews by owner: $owner") =>  
    rep.getReviewBy(ownerCriteria(owner))  
    ...  
  override def getReviewBy(rate: Double): F[Seq[Review]] =  
    Logger[F].info("Getting reviews by rate: $rate") =>  
    rep.getReviewBy(rateCriteria(rate))  
    ...  
  override def getReviewBy(owner: String, rate: Double): F[Seq[Review]] =  
    Logger[F].info("Getting reviews by owner: $owner_and_rate: $rate") =>  
    rep.getReviewBy(ownerCriteria(owner) and rateCriteria(rate))  
  }  
}
```

```
private def ownerCriteria[D <: CriteriaTag: EQ: Sym](owner: String): Criteria[D] =  
  col[D]("owner") === lit(owner)  
  
private def rateCriteria[D <: CriteriaTag: GT: Sym](rate: Double): Criteria[D] =  
  col[D]("rate") gt lit(rate)
```



RESULTADOS

→ Escala

→ Mantenemos lógica de negocio

```
/** MongoDB ReviewServices implementation */  
def using[F[_]: Async: Logger](mongo: MongoClient): ReviewServices[F] =  
  build[F, MongoDB](ReviewRepository.using(mongo))  
  
/** JDBC-PostgreSQL ReviewServices implementation */  
def using[F[_]: Async: Logger](jdbc: Connection): ReviewServices[F] =  
  build[F, Postgres](ReviewRepository.using(jdbc))
```

Criteria4s - Resultados

Respondiendo a CAPA 2 = Repository (II)

PROBLEMA

→ No escala

```
trait ReviewRepository_[F[_]] {  
  def getReviews: F[Seq[Review]]  
  
  def getReviewByOwner(owner: String): F[Seq[Review]]  
  
  def getReviewByRate(rate: Double): F[Seq[Review]]  
  
  def getReviewByOwnerAndRate(owner: String, rate: Double): F[Seq[Review]]  
  
  def getReviewByAddress(address: String): F[Seq[Review]]  
  
  def getReviewByRateAndAddress(rate: Double, address: String): F[Seq[Review]]  
  
  def getReviewByOwnerAndAddress(owner: String, address: String): F[Seq[Review]]  
  
  def getReviewByOwnerRateAndAddress(owner: String, rate: Double, address: String): F[Seq[Review]]  
  
  def getReviewByComment(comment: String): F[Seq[Review]]  
  
  def getReviewByRateAndComment(rate: Double, comment: String): F[Seq[Review]]  
  
  def getReviewByOwnerAndComment(owner: String, comment: String): F[Seq[Review]]  
  
  def getReviewByOwnerRateAndComment(owner: String, rate: Double, comment: String): F[Seq[Review]]  
  
  def getReviewByAddressAndComment(address: String, comment: String): F[Seq[Review]]  
}
```



RESULTADOS

- Escala
- Mantenemos lógica de negocio
- Prever lógica problemática

```
def otherCriteria[D <: CriteriaTag: GT: Sym: ISNOTNULL: AND](rate: Double): Criteria[D] =  
  col[D]("rate") gt lit(rate) and col[D]("owner").isNotNull
```

Diagram illustrating the evolution from a generic repository trait to specific MongoDB and JDBC implementations. Arrows show the flow from the trait to the implementation code.

```
/** Generic ReviewServices implementation */  
def build[F[_]: Async: Logger, D <: CriteriaTag: EQ: GT: Sym: AND: ISNOTNULL: ](  
  rep: ReviewRepository[F]  
): ReviewServices[F] =  
  new ReviewServices[F] {  
  
    /* ... */  
  
    override def getReviewBy(rate: Double): F[Seq[Review]] =  
      Logger[F].info(s"Getting reviews by rate: $rate") *>  
      rep.getReviewBy(otherCriteria(rate))  
  }  
  
/* MongoDB ReviewServices implementation */  
def using[F[_]: Async: Logger](mongo: MongoClient) =  
  build[F, MongoClient](ReviewRepository.using(mongo))  
  
/* JDBC-PostgreSQL ReviewServices implementation */  
def using[F[_]: Async: Logger](jdbc: Connection): ReviewServices[F] =  
  build[F, Postgres](ReviewRepository.using(jdbc))
```

Criteria4s

Ahora
*Nuestra carta
de bebidas*

Lo que tenemos - Release 0.8+

- Soporte para colecciones
- Soporte para rangos
- Uso de type class **SHOW** para la representación (*antes Sym*)

```
trait Collection[D <: CriteriaTag, +V] extends Ref[D, Seq[V]]  
trait Range[D <: CriteriaTag, +V]      extends Ref[D, (V, V)]  
  
trait Show[-V, D <: CriteriaTag] {  
  def show(v: V): String  
}  
  
implicit val showColumn: Show[Column, Postgres] = Show.create(col => s"${col.colName}`")  
  
implicit def showValue[V]: Show[V, Postgres] = Show.create(v => s"'$v'")  
  
implicit def showSeq[T](implicit show: Show[T, Postgres]): Show[Seq[T], Postgres] =  
  Show.create(_.map(show.show).mkString("(, , , , )"))  
  
implicit def betweenShow[T](implicit show: Show[T, Postgres]): Show[(T, T), Postgres] =  
  Show.create { case (l, r) => s"${show.show(l)} TO ${show.show(r)}" }
```

Criteria4s

Ahora
*Nuestra carta
de bebidas*

Lo que tenemos - Release 0.8+

- Soporte para colecciones
- Soporte para rangos
- Uso de type class **SHOW** para la representación (*antes Sym*)
- Uso **Kind Projection** para simplificar *type lambdas expressions*
- Bug fixing para **BETWEEN** y **NOTBETWEEN**

```
def boilerplatedExpr[T <: CriteriaTag: BETWEEN:  
{ type A[D <: CriteriaTag] = Show[Column, D] }#A:  
{ type A[D <: CriteriaTag] = Show[(Int, Int), D] }#A]: Criteria[T] =  
  col[T]("column") between range(100, 150)  
  
def expr[T <: CriteriaTag: BETWEEN: Show[Column, *]: Show[(Int, Int), *]]: Criteria[T] =  
  col[T]("column") between range(100, 150)
```

```
col[Postgres]("a") in array(1, 2, 3)  
'a` IN ('1', '2', '3')
```

```
col[Postgres]("b") notBetween range("A", "Z")  
'b` NOT BETWEEN 'A' TO 'Z'
```

Criteria4s

Ahora Nuestra carta de bebidas

Lo que tenemos - Release 0.8+

→ 📣 Publicación en **ScalaTimes**

→ 16 ⭐ + 1 👏 contributor + 1 ⚡ fork + 1 🛡 pull request

The screenshot shows the ScalaTimes website with the following details:

- Header:** A free, once-weekly Scala news flash. Easy to unsubscribe. Goes out every Thursday.
- Section:** SCALA TIMES WEEKLY SCALA NEWSPAPER
- Search Bar:** Search...
- Date:** March 7th, 2024, ISSUE 5
- Content:**
 - Scala Times Issue #528**: Jacek Kunički - My Scala Story. Kalix tutorial: Building invoke application. Safe direct-style Scala: Ox 0.1.0 released. LambdaConf - Call for Papers, May 4th-10th, Estes Park, Colorado. LambdaConf - The Grand Hackathon Finale, May 10th.
 - Releases**: Cats Effect 3.5.4
 - caliban 2.5.3**: This release includes support for subscriptions over Server-Sent Events, as well as a few bug fixes and improvements.
 - criteria4s**: A simple domain-specific language (DSL) to define criteria and predicate expressions for any data stores by using Scala type class mechanisms in a type-safe way. It pretends to be agnostic to any data store and it is extensible to support any kind of data stores.
 - Tapir 1.9.11**: Includes quite a bunch of performance improvements and dependency updates.
- Footer:** E-mail address, Subscribe button, Scala Times Issue #514, Scala Times Issue #513, Scala 2 Macro Tutorial, A Beginner's Guide to GraphQL In.

Criteria4s

Futuro

Próximas mezclas

Hacía dónde vamos - **Under construction**

- **Issue #8** - Complete extensions of predicate functions by using boolean symbols
- **Issue # 7** - Complete SQL predicate expression
- **Issue # 18** - Add Spark SQL expression interpreter
- **Issue # 19** - LIMIT and OFFSET criteria expressions (API)
- Type checker para operaciones entre valores:
 - 3 > "hola"
 - ⇒ 3 >[Int] 4
 - ⇒ 3 >[Any] "hola"
- 4.contains(1,2,3,4)
- ¿Qué hacemos con las columnas?
- Añadir más intérpretes para SQL / NoSQL
 - **Issue # 14** - Add MongoDB expression interpreter
 - **Issue # 21** - Add Elasticsearch expression interpreter
 - **Issue # 20** - Add PostgreSQL expression interpreter



Preguntas

Referencias

- **Types and programming languages.** *Benjamin C. Pierce*
- **Typed Tagless Final Interpreters.** *Oleg Kiselyov ([Paper](#))*
- **Tagless-final o la abstracción sin culpa.** *Habla ([Yotube](#)).*
- **Semantics with Applications.** *Hanne Riis Nielson, Flemming Nielson.*

$$S \xrightarrow{f} T$$

$$G \xrightarrow{\varphi} H$$

$$X \xrightarrow{g} Y$$

Criteria4s

Filtros, filtros everywhere

Rafael Fernández Ortiz