Define your own rules:

dynamic system behavior control at runtime

...

About me

Veronika Yastrebova

Backend Scala Team Leader

at Appodeal

BidMachine

Do not worry about Scala 😄

- Software Design

Do not worry about Scala 😄

- Software Design
- Functional programming
  - Higher-order functions
  - Pattern matching
  - Functors

Example

Discount for the USA users on Black Friday

```
if (country == "US" && today.equals(BLACK_FRIDAY))
  discountPrice
else
  price
```

+ only for specific SDK versions

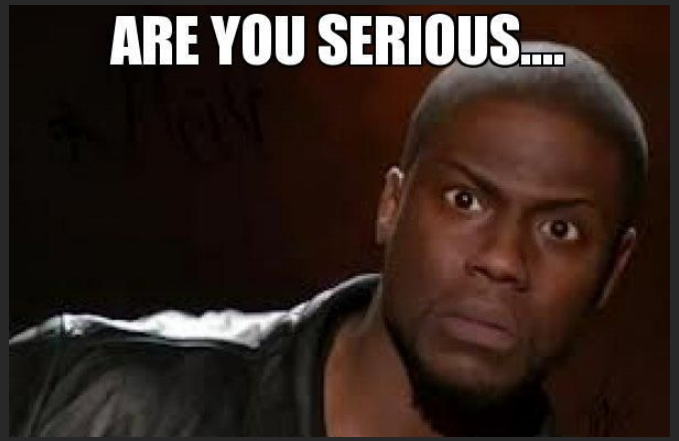
```
if (country == "US" && today.equals(BLACK_FRIDAY) && (os == OS.iOS && sdkVer ≥ "2.3.0.2") || (os == OS.Android && sdkVer ≥ "11.10.1"))
  discountPrice
else
  price
```

+ enable for test devices

+ non-technical users can control the feature

+ ...

```
if (country == "US" && today.equals(BLACK_FRIDAY) && (os == OS.iOS && sdkVer ≥ "2.3.0.2") || (os == OS.Android && sdkVer ≥ "11.10.1") && orderedItems > 20 && devi
  discountPrice
else
  price
```



Will better code style really help?

```
val clientWhitelist = seller.configuration.clientWhitelist
val sdkVer          = new ComparableVersion(ctx.sdkver)
val `1.7.1`         = new ComparableVersion( version = "1.7.1")
val osCheck         = ctx.isEnabledOnIos || (ctx.isEnabledOnAndroid && sdkVer.compareTo(`1.7.1`) > 0)
val countries       = List("RUS", "ARM", "AZE", "BLR", "GEO", "KAZ", "KGZ", "MDA", "TJK", "UKR", "UZB")
val country         = ctx.geo.flatMap(_.country).exists(countries.contains)
val isWhitelisted   = clientWhitelist.isEmpty || clientWhitelist.contains(ctx.bundle)

(if (seller.configuration.clientSdkEnabled && isWhitelisted && osCheck)
```

* real example from our app

Problem definition

Manage changing system behavior in real-time in a manager-friendly way

Problem definition

Manage changing system behavior in real-time in a manager-friendly way

Problem definition

Manage changing system behavior in real-time in a manager-friendly way

Problem definition

Manage changing system behavior in real-time in a manager-friendly way

Solution

Rule matcher

Solution

Rule matcher

- verify rule validity
- keep rules easy to read and manage
- use any app attribute

What is an attribute?

Value of any property

What is an attribute?

Value of any property

User

user id

device os

country

phone number

What is an attribute?

Value of any property

User

user id

device os

country

phone number

System

day

hour

year

What is an attribute?

Value of any property

User

user id

device os

country

phone number

System

day

hour

year

Historical values

orders amount

item price

last activity time

20$

```
country  ==  US
os       ==  ios
```

20$

country == US
os == ios

20$

country == DE
os == android

20$

country == US
os == ios

20$

country == US
os == ios

20$

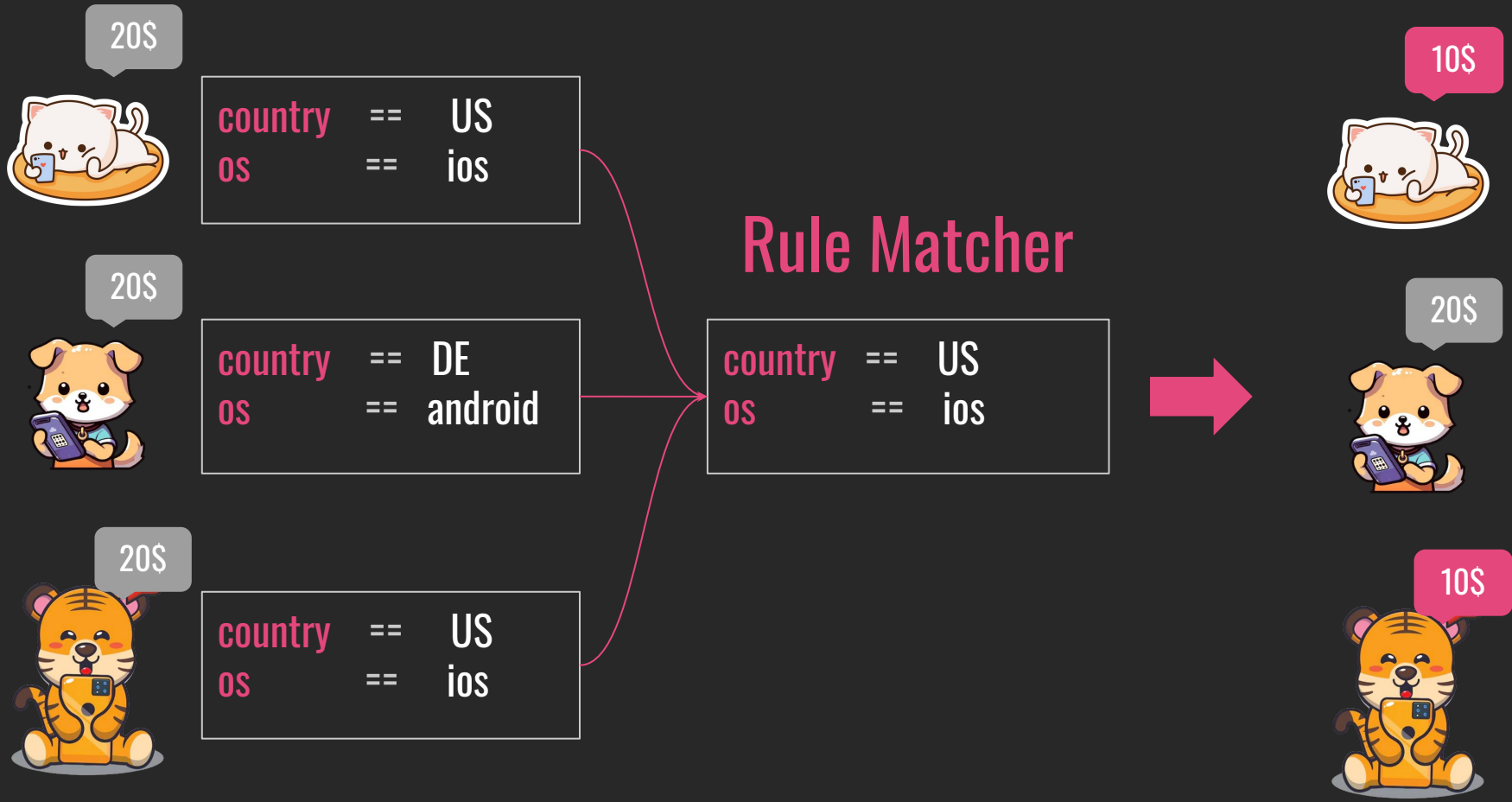
country == DE
os == android

20$

country == US
os == ios

Rule Matcher

country == US
os == ios


20$
country == US
os == ios
20$
country == DE
os == android
20$
country == US
os == ios
Rule Matcher
country == US
os == ios
10$
20$
10$

App

```
val allowedOnIOS = os == OS.iOS && sdkVer >= "2.3.0.2"
val allowedOnAndroid = os == OS.Android && sdkVer >= "11.10.1"

if (country == "US" && today.equals(BLACK_FRIDAY) && allowedOnIOS || allowedOnAndroid)
  discountPrice
else
  price
```

Rule system

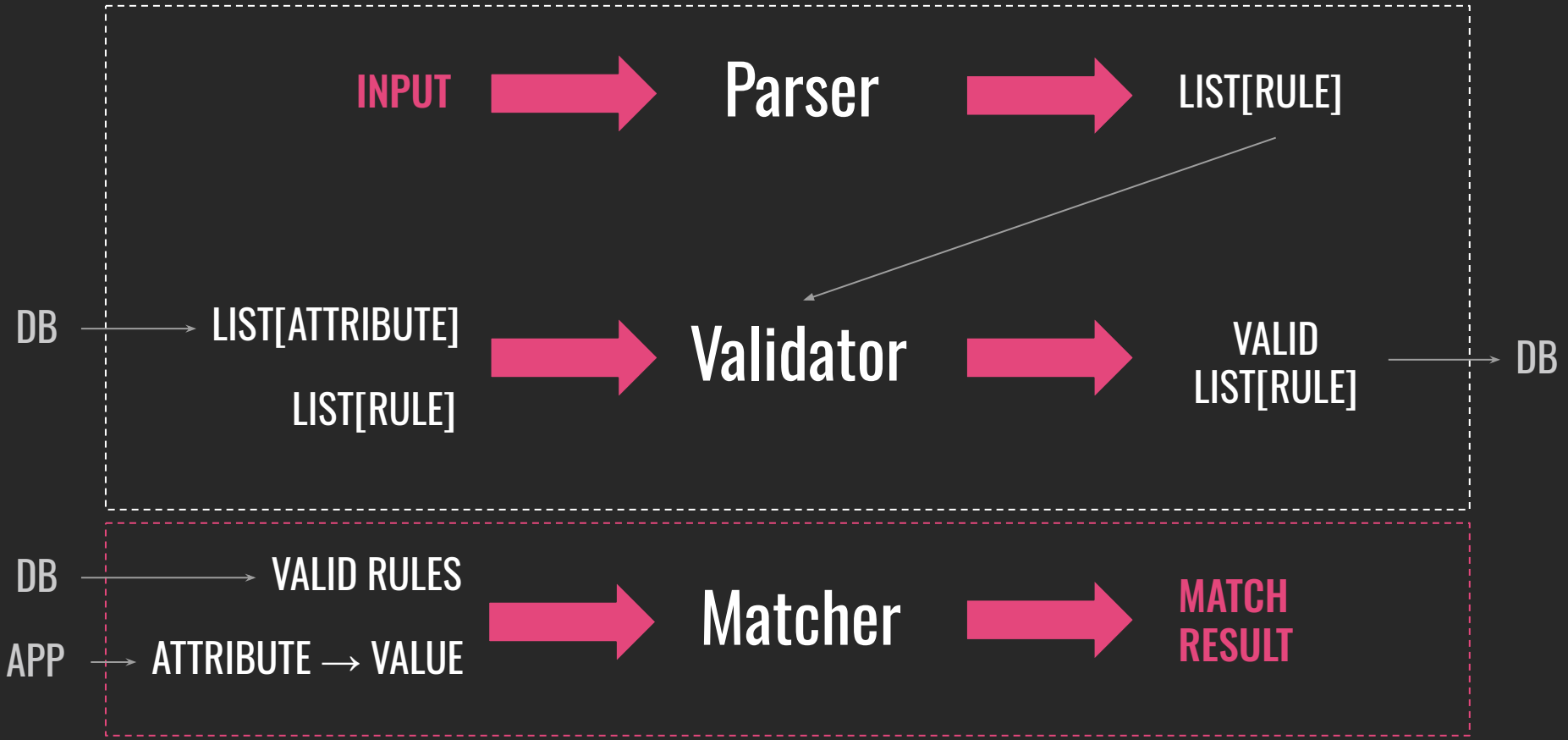
```
country == "US"
os
  == "android"
    sdkver >= "11.10.1"
  == "ios"
    sdkver >= "2.3.0.2"
```

App

```
val matchResult = ruleMatcher(attributes)
if (matchResult)
  discountPrice
else
  price
```

What's next?

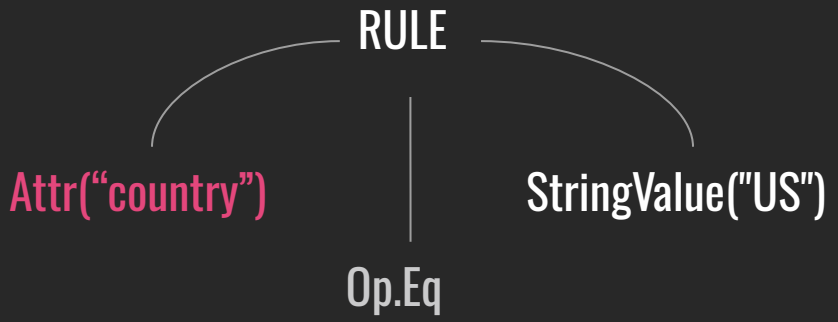
- Architecture
- Domain models
- Recursion schemes


INPUT
Parser
LIST[RULE]
DB
LIST[ATTRIBUTE]
LIST[RULE]
Validator
VALID
LIST[RULE]
DB
DB
VALID RULES
APP
ATTRIBUTE → VALUE
Matcher
MATCH
RESULT


STRING
Parser
LIST [RULE]

country == "US"

attribute operation value



Example domain models

```scala
sealed abstract class Op(val value: String)
  extends StringEnumEntry with Product with Serializable

object Op extends StringEnum[Op] {

  case object Eq    extends Op( value = "==")
  case object NotEq extends Op( value = "≠")
  case object Lt    extends Op( value = "<")
  case object LtE   extends Op( value = "⩽")
  case object Gt    extends Op( value = ">")
  case object GtE   extends Op( value = "⩾")
  case object In    extends Op( value = "in")
  case object NotIn extends Op( value = "not in")
```

Example domain models

```
sealed abstract class Value extends Serializable with Product

object Value {

  case class BoolValue(value: Boolean)         extends Value
  case class NumericValue(value: Double)       extends Value
  case class StringValue(value: String)        extends Value
  case class ArrayValue(value: Vector[Value]) extends Value
  case object NoneValue                        extends Value

}
```

```
case class Rule(attr: Attr, op: Op, value: Value)
```

Example parsing

```scala
private def op[O <: Op, T: P](op: O)            = lit(op.value, op)
private def lt[T: P]                            = op(Op.Lt)
private def gt[T: P]                            = op(Op.Gt)
private def eq[T: P]                            = op(Op.Eq)
private def gte[T: P]                           = op(Op.GtE)
private def lte[T: P]                           = op(Op.LtE)
private def notEq[T: P]                         = op(Op.NotEq)
private def in[T: P]                            = lit(lex.kw("in"), Op.In)
private def notIn[T: P]                         = lit(lex.kw("not") ~ lex.kw("in"), Op.NotIn)
```

* using fastparse library


LIST[ATTRIBUTE]
LIST[RULE]
Validator
VALID
LIST[RULE]

Validate

- operations
- allowed values

country == "US"

attribute	country
operation	Eq
value	StringValue("US")

RULE

id	country
type	String
allowed values	US, EN, BR, ...

ATTR

Validate operations

Attribute type

Rule value type

List of allowed operations

If Rule operation is in this list

```
private val equalityOps   = Set(Eq, NotEq)
private val comparisonOps = Set(Eq, NotEq, Lt, LtE, Gt, GtE)
private val setOps        = Set(In, NotIn)

private val allowedOps: (AttributeType, Value) ⇒ Set[Op] = {
  case (Numeric, _: NumericValue)                                          ⇒ comparisonOps
  case (Numeric | String, _: ArrayValue)                                   ⇒ setOps
  case (String, _: StringValue)                                            ⇒ equalityOps
  case (Numeric, _: NumericValue) | (Bool, _: BoolValue) | (Array, _: ArrayValue) ⇒ equalityOps
  case _                                                                   ⇒ Set.empty
}
```

String

"US"

Op.Eq, Op.NotEq
(allowed ops)

Op.Eq is in
the list

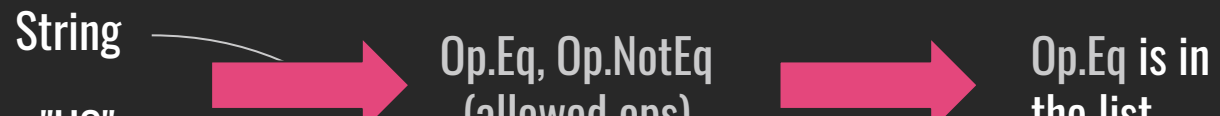
country == "US"

attribute	country
operation	Eq
value	StringValue("US")

RULE

id	country
type	String
allowed values	US, EN, BR, ...

ATTR



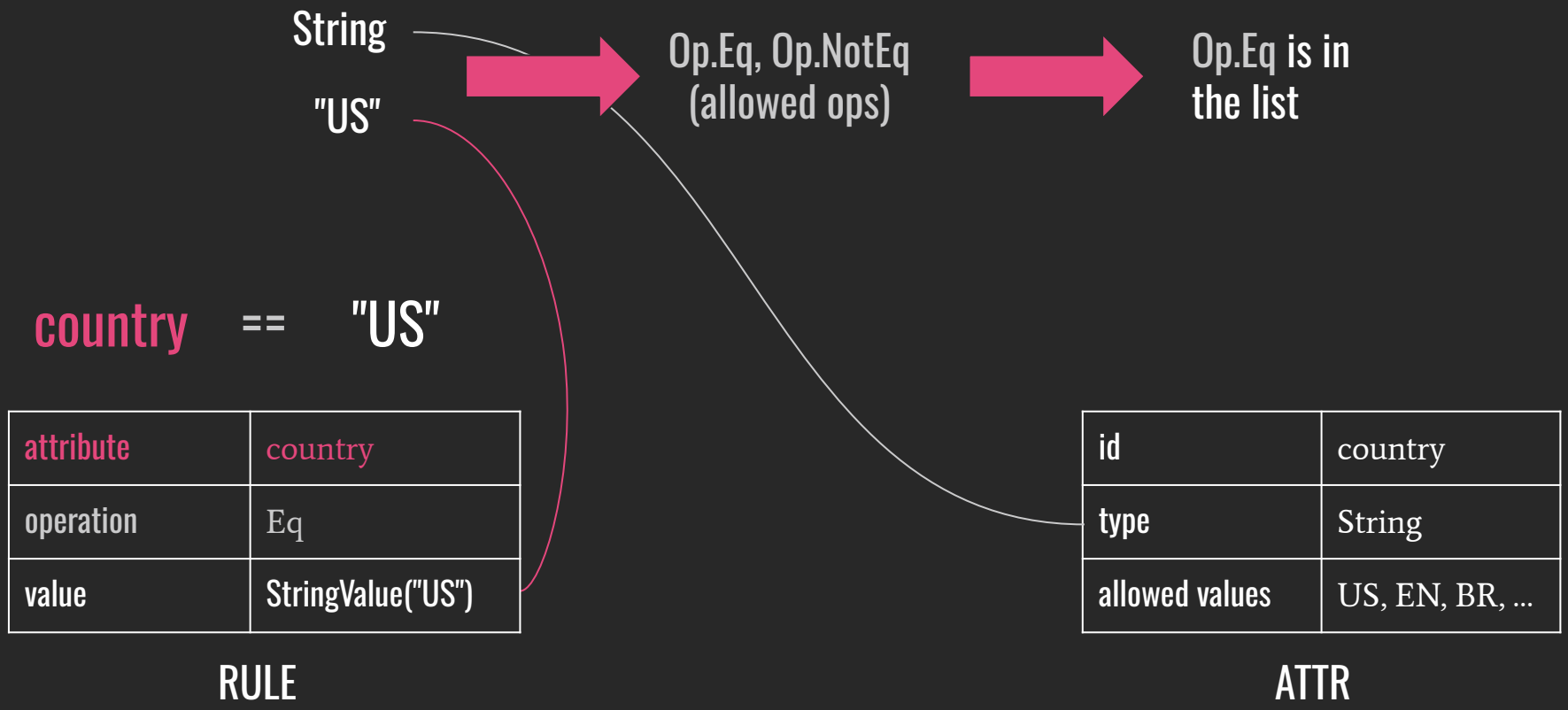
country == "US"

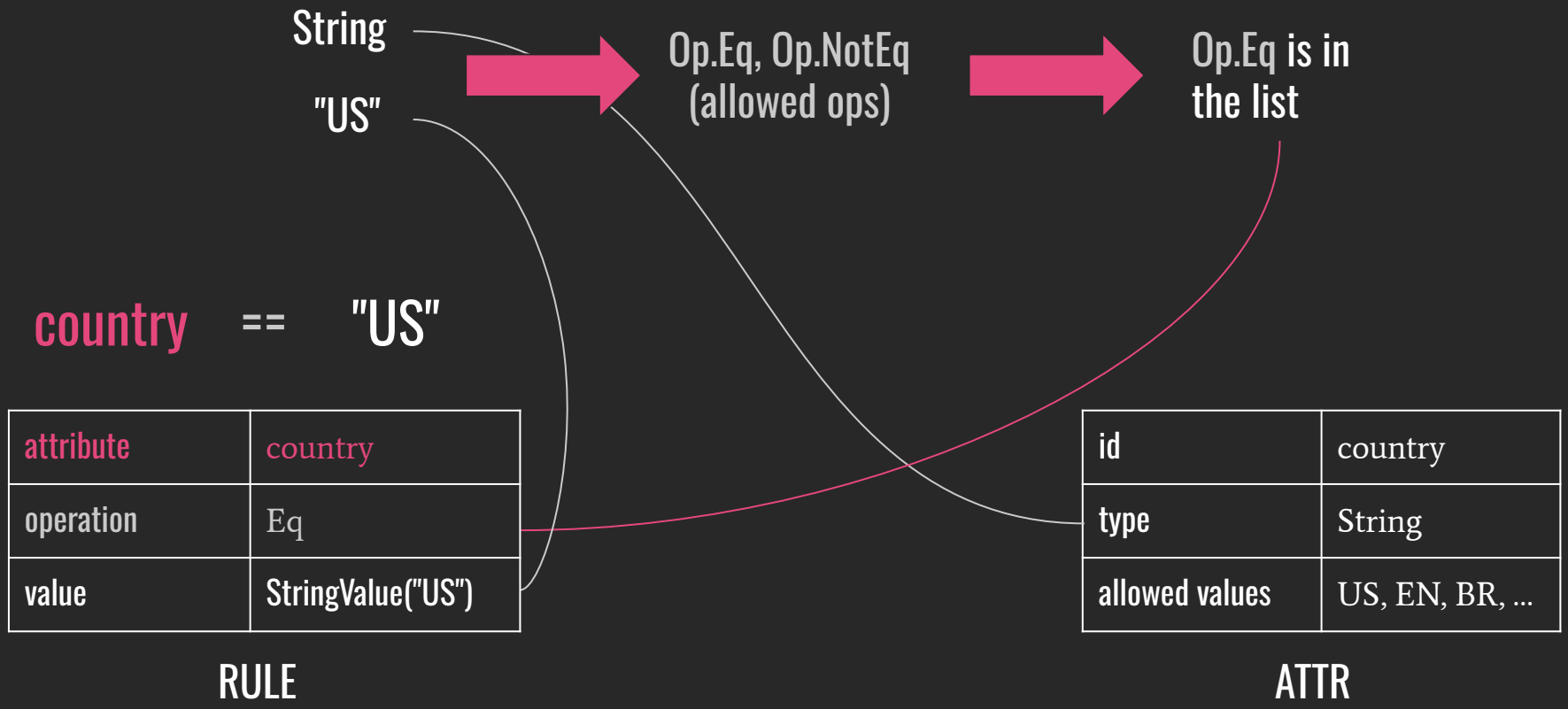
attribute	country
operation	Eq
value	StringValue("US")

RULE

id	country
type	String
allowed values	US, EN, BR, ...

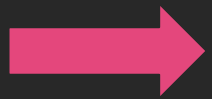
ATTR


String
"US"
Op.Eq, Op.NotEq
(allowed ops)
Op.Eq is in
the list
country == "US"
attribute
country
operation
Eq
value
StringValue("US")
RULE
id
country
type
String
allowed values
US, EN, BR, ...
ATTR


String
"US"
Op.Eq, Op.NotEq
(allowed ops)
Op.Eq is in
the list
country == "US"
attribute
country
operation
Eq
value
StringValue("US")
RULE
id
country
type
String
allowed values
US, EN, BR, ...
ATTR

INPUT

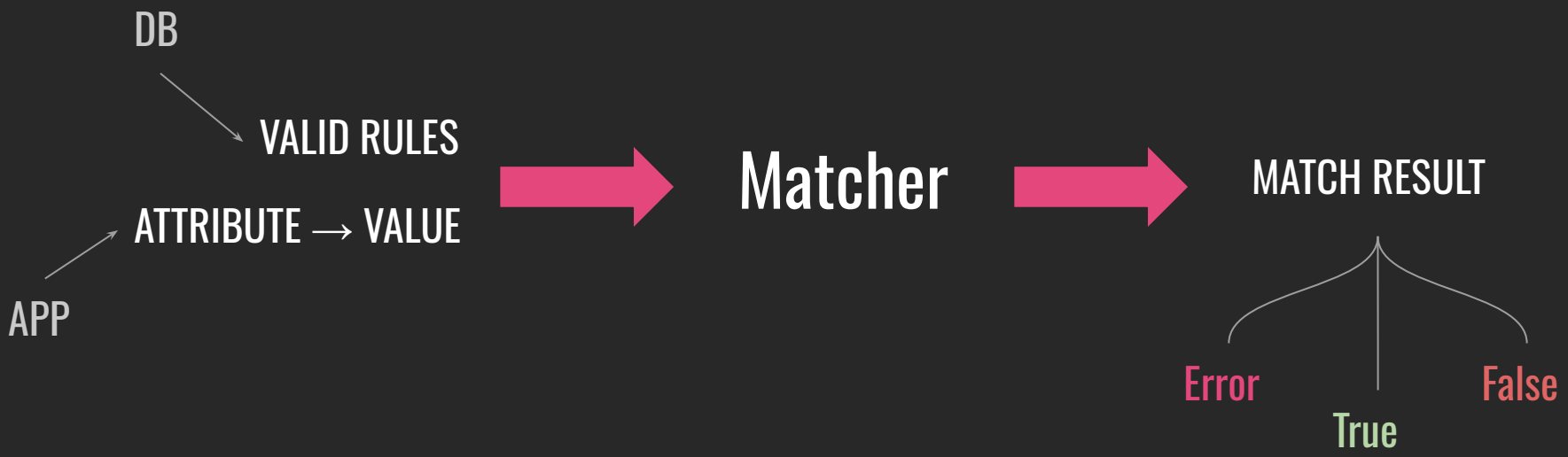
```
country == "US"
os
  == "android"
    sdkver >= "11.10.1"
  == "ios"
    sdkver >= "2.3.0.2"
```



VALID LIST[RULE]

```
List(
  Rule(Attr("country"), Op.Eq, StringValue("US")),
  Rule(Attr("os"), Op.Eq, StringValue("android")),
  Rule(Attr("sdkver"), Op.GtE, StringValue("11.10.1")),
  Rule(Attr("os"), Op.Eq, StringValue("ios")),
  Rule(Attr("sdkver"), Op.GtE, StringValue("2.3.0.2"))
)
```

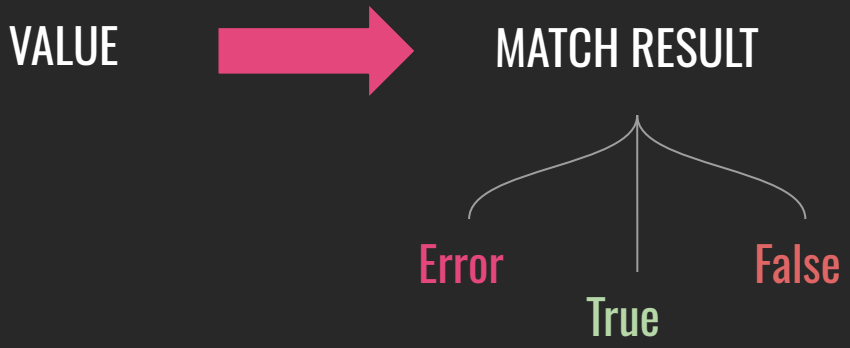

DB


DB
VALID RULES
ATTRIBUTE → VALUE
APP
Matcher
MATCH RESULT
Error
True
False

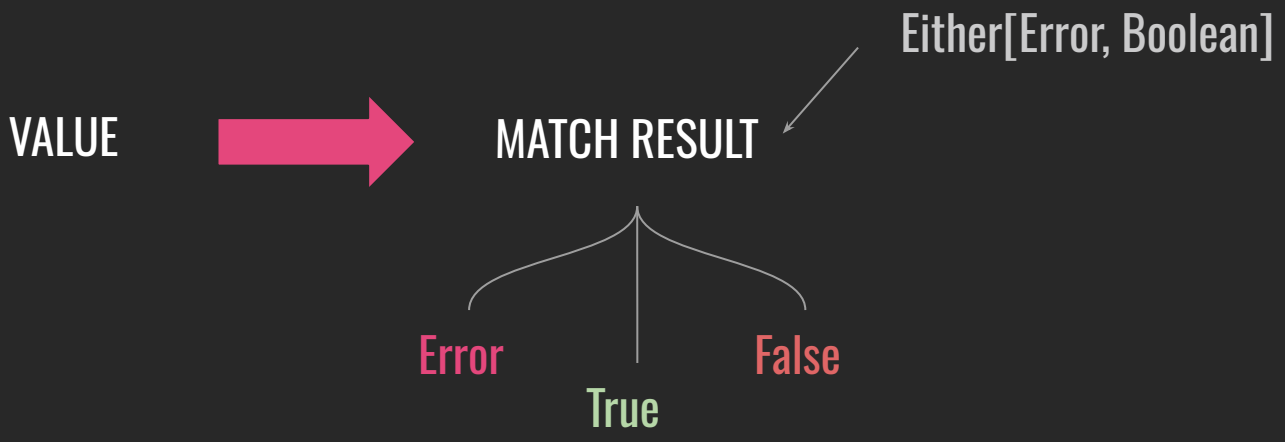
Example passing attributes

```
def attributesFrom(request: Request): Attr ⇒ Option[Value] = {

  val userAttributes: Map[Attr, Option[Value]] = Map(
    Attributes.Os → request.device.os.map(Value.StringValue),
    Attributes.Country → request.device.geo.flatMap(_.country.map(Value.StringValue)),
    Attributes.Hour → Value.NumericValue(now.getHour.toDouble).some
  )

  attr ⇒ userAttributes.get(attr).flatten
}
```

Predicate



Predicate



Simple Predicate

```
case object TruePred extends Predicate {
  override def apply(v1: Value): Matching = Right(true)
}

case object FalsePred extends Predicate {
  override def apply(v1: Value): Matching = Right(false)
}
```

Rule Predicate

toString = "P(_ $op $value)"

country == "US"  P(_ Eq StringValue(US))

Rule Predicate

```scala
case class NumGtE(rightSideValue: NumericValue) extends Predicate {
  override def apply(leftSideValue: Value): Matching = leftSideValue match {
    case NumericValue(leftSideValue) ⇒ Right(leftSideValue ⩾ rightSideValue.value)
    case other ⇒ Left(s"$other is not a Numeric")
  }
}

case class StrEq(rightSideValue: StringValue) extends Predicate {
  override def apply(leftSideValue: Value): Matching = leftSideValue match {
    case StringValue(leftSideValue) ⇒ Right(leftSideValue == rightSideValue.value)
    case other ⇒ Left(s"$other is not a String")
  }
}
```

Rules composition

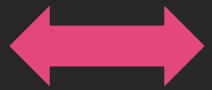
os == "android"
sdkver >= "11.10.1"
country == "US"

os == "android" &&
sdkver >= "11.10.1" &&
country == "US"

```
List(
  StrEq(StringValue("android")),
  VerGtE(StringValue("11.10.1")),
  StrEq(StringValue("US"))
)
```

Rules composition

```
os
  == "android"
     sdkver >= "11.10.1"
  == "ios"
     sdkver >= "2.3.0.2"
```



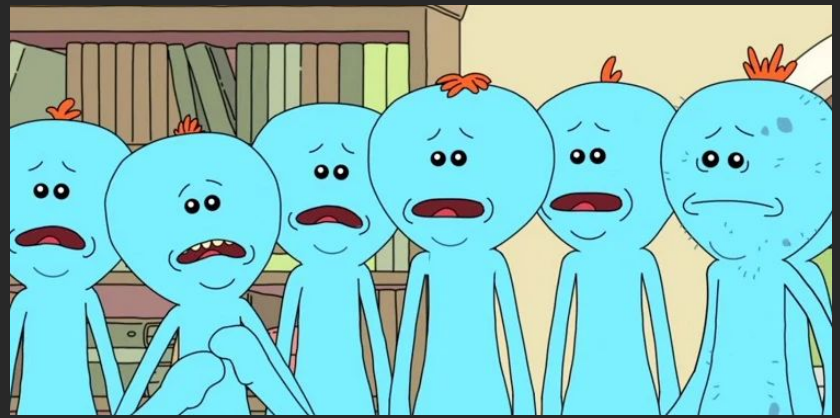
```
(os == "android" &&
 sdkver >= "11.10.1") ||
 (os == "ios" &&
 sdkver >= "2.3.0.2")
```

We need children

```
case class Rule(attr: At  p: Op, value: Value, children: List[Rule]) {

  def append(rules: List[Rule]): Rule = copy(children = this.children ++ rules)

}
```

```
val rules = List(
  Rule(
    Attr("os"),
    Op.Eq,
    Value.StringValue("android"),
    children = List(Rule(Attr("sdkver"), Op.GtE, Value.StringValue("11.10.1"), children = List.empty))
  ),
  Rule(
    Attr("os"),
    Op.Eq,
    Value.StringValue("ios"),
    children = List(Rule(Attr("sdkver"), Op.GtE, Value.StringValue("2.3.0.2"), children = List.empty))
  )
)

val predicates: List[Predicate] = ???
```

Recursion



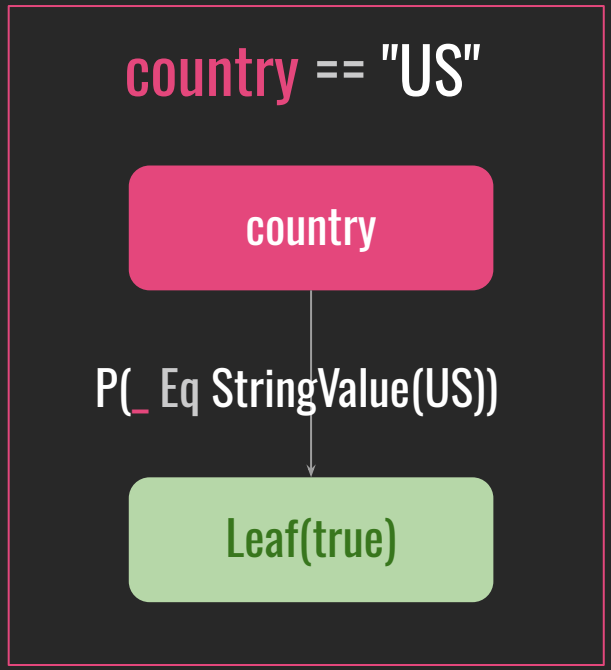
Matching flow

- Build predicate tree from list of Rules
- Recursively evaluate predicates for incoming attributes
- Collapse to matching result

Recursive Tree

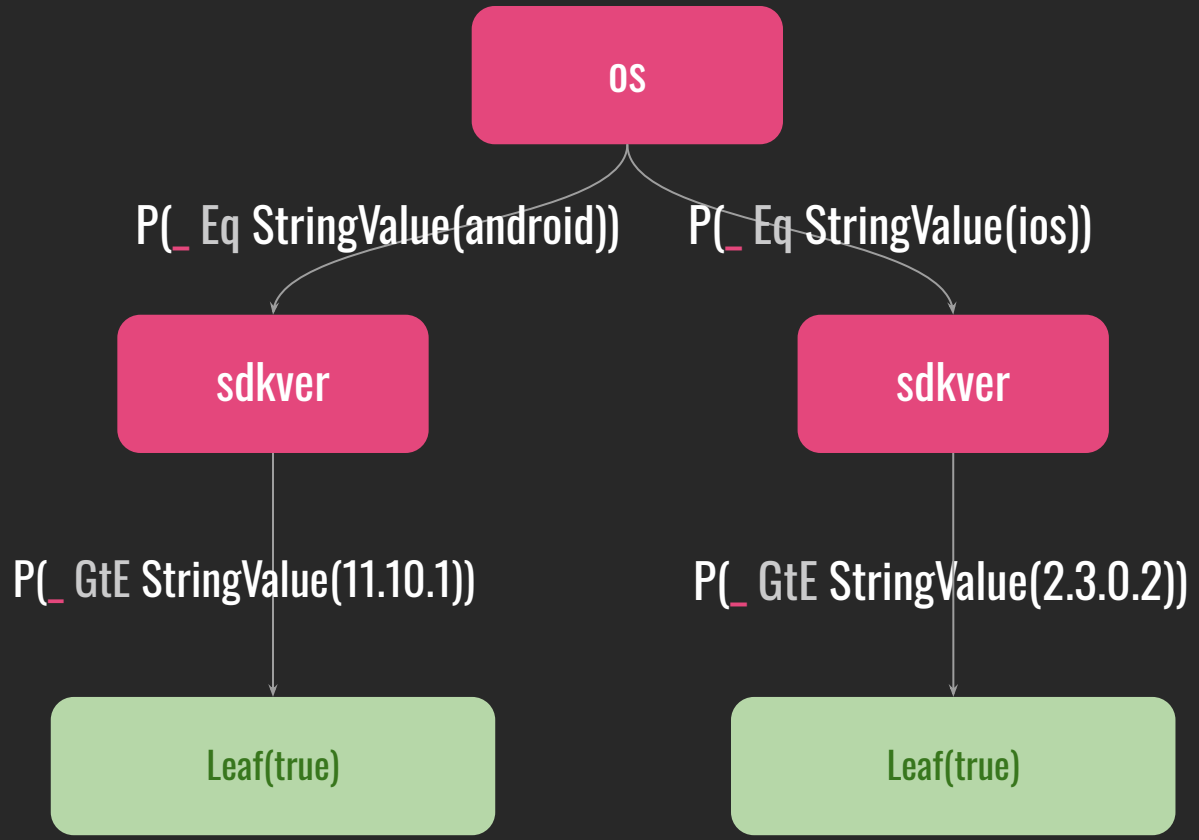
```
sealed trait TreeR

case class Leaf(p: Boolean) extends TreeR
case class Node(attr: Attr, children: NonEmptyList[(Predicate, TreeR)]) extends TreeR
```

```scala
sealed trait TreeR

case class Leaf(p: Boolean) extends TreeR
case class Node(attr: Attr, children: NonEmptyList[(Predicate, TreeR)]) extends TreeR
```




```scala
Node(
  Attr("country"),
  NonEmptyList.one(
    (
      P(_ Eq StringValue("US")),
      Leaf(true)
    )
  )
)
```

```
os
  == "android"
     sdkver >= "11.10.1"
  == "ios"
     sdkver >= "2.3.0.2"
```



Build TreeR

```
type AttrInfoProvider = Attr ⇒ Option[AttrInfo]

def build(attrInfoProvider: AttrInfoProvider)(rules: List[Rule]): TreeR = {
  def mkPred(rule: Rule): (Predicate, TreeR) =
    (Predicate.fromRule(rule)(attrInfoProvider), build(attrInfoProvider)(rule.children))

  rules match {
    case Nil          ⇒ Leaf(true)
    case head :: Nil ⇒ Node(head.attr, NonEmptyList.one(mkPred(head)))

    case rules ⇒
      val attr           = rules.head.attr
      val (root, inner) = rules.span(_.attr == attr)
      val predicates     = root.map(_.append(inner)).map(mkPred)

      Node(attr, NonEmptyList.fromListUnsafe(predicates))
  }
}
```

Fold TreeR

```scala
type Matching = Either[MatchingFailure, Boolean]

object Matching {
  def failure(reason: String): Matching = Left(reason)

  def success(b: Boolean): Matching = Right(b)
}

def fold(getAttrValue: Attr ⇒ Option[Value]): TreeR ⇒ Matching = {
  case Leaf(p)              ⇒ Matching.success(p)
  case Node(attr, children) ⇒
    val step = {
      for {
        matching ← children.findM { case (predicate, _) ⇒ predicate(getAttrValue(attr).getOrElse(NoneValue)) }
        (_, sub) ← matching.toRight(s"No matching predicate found")
      } yield sub
    } leftMap Matching.failure

    step match {
      case Left(matching) ⇒ matching
      case Right(subtree) ⇒ fold(getAttrValue)(subtree)
    }
}
```

Matcher

```
type Matching         = Either[MatchingFailure, Boolean]
type AttrInfoProvider = Attr ⇒ Option[AttrInfo]
type Sample           = Attr ⇒ Option[Value]

def matcher(attrInfoProvider: AttrInfoProvider)(rules: List[Rule]): Sample ⇒ Matching = {
  val tree = TreeR.build(attrInfoProvider)(rules)

  sample ⇒ TreeR.fold(sample)(tree)
}
```

Can we hide the recursion?

All we need

- Fix
- Higher kinded Tree
- Functor

Fixed point

```
// the fixed point of a function is f(x) = x
case class Fix[F[_]](unFix: F[Fix[F]])
```

f(x) = x^2 has a fixed point of 1 since $1^2 = 1$.

A higher-kinded TreeF

```scala
sealed trait TreeR

case class Leaf(p: Boolean) extends TreeR
case class Node(attr: Attr, children: NonEmptyList[(Predicate, TreeR)]) extends TreeR
```

```scala
sealed trait TreeF[A]

case class Leaf[A](p: Boolean)                                    extends TreeF[A]
case class Node[A](attr: Attr, children: NonEmptyList[(Predicate, A)]) extends TreeF[A]
```

```scala
// TreeF[Fix[TreeF]] (because we recapture the recursion and we made A = Fix[TreeF])
case class Fix[F[_]](unFix: F[Fix[F]])
```

Functor

```
implicit val treeFunctor: Functor[TreeF] = new Functor[TreeF] {
  def map[A, B](fa: TreeF[A])(f: A ⇒ B): TreeF[B] =
    fa match {
      case Leaf(p)              ⇒ Leaf(p) // terminating case for the recursion, without type param A
      case Node(attr, children) ⇒ Node(attr, children.map(_.map(f)))
    }
}
```

MATRYOSHKA

Recursion schemes

ANA (anamorphism) == unfold

```
def ana[F[_] : Functor, A](f: A ⇒ F[A]): A ⇒ Fix[F] =
  a ⇒ Fix(f(a) map ana(f))
```

MATRYOSHKA

Recursion schemes

ANA (anamorphism) == unfold

```
def ana[F[_] : Functor, A](f: A ⇒ F[A]): A ⇒ Fix[F] =
  a ⇒ Fix(f(a) map ana(f))
```

CATA (catamorphism) == fold

```
def cata[F[_] : Functor, A](f: F[A] ⇒ A): Fix[F] ⇒ A =
  fix ⇒ f(fix.unfix map cata(f))
```

MATRYOSHKA

Recursion schemes

ANA (anamorphism) == unfold

```
def ana[F[_] : Functor, A](f: A ⇒ F[A]): A ⇒ Fix[F] =
  a ⇒ Fix(f(a) map ana(f))
```

CATA (catamorphism) == fold

```
def cata[F[_] : Functor, A](f: F[A] ⇒ A): Fix[F] ⇒ A =
  fix ⇒ f(fix.unfix map cata(f))
```

HYLO (hylomorphism) == ANA + CATA

```
def hylo[F[_] : Functor, A, B](f: F[B] ⇒ B)(g: A ⇒ F[A]): A ⇒ B =
  a ⇒ f(g(a) map hylo(f)(g))
```

New matching flow

- Build predicate tree from list of Rules
- Find Path in Tree for incoming attributes
- Follow the Path and collapse to matching result

```scala
def build(attrInfoProvider: AttrInfoProvider)(rules: List[Rule]): TreeR = {
  def mkPred(rule: Rule): (Predicate, TreeR) =
    (Predicate.fromRule(rule)(attrInfoProvider), build(attrInfoProvider)(rule.children))

  rules match {
    case Nil         ⇒ Leaf(true)
    case head :: Nil ⇒ Node(head.attr, NonEmptyList.one(mkPred(head)))

    case rules ⇒
      val attr          = rules.head.attr
      val (root, inner) = rules.span(_.attr == attr)
      val predicates    = root.map(_.append(inner)).map(mkPred)

      Node(attr, NonEmptyList.fromListUnsafe(predicates))
  }
}
```

```scala
def build(attrInfoProvider: AttrInfoProvider): Coalgebra[TreeF, List[Rule]] = {
  def mkPred(rule: Rule): (Predicate, List[Rule]) =
    (Predicate.fromRule(rule)(attrInfoProvider), rule.children)

  // List[Rule] ⇒ TreeF[List[Rule]]
  // A ⇒ F[A], where F -- functor
  Coalgebra {
    case Nil         ⇒ Leaf(true)
    case head :: Nil ⇒ Node(head.attr, NonEmptyList.one(mkPred(head)))

    case rules ⇒
      val attr = rules.head.attr
      val (root, inner) = rules.span(_.attr == attr)
      val predicates = root.map(_.append(inner)).map(mkPred)

      Node(attr, NonEmptyList.fromListUnsafe(predicates))
  }
}
```

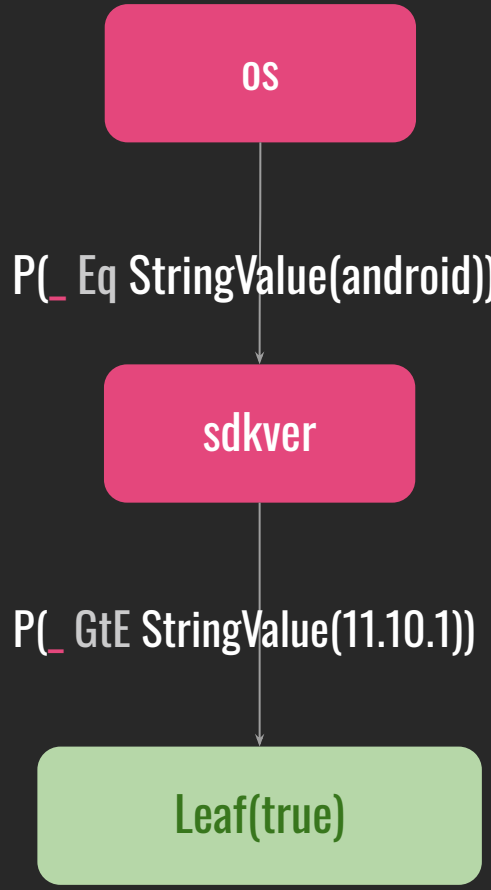
```scala
// ana(f: A ⇒ F[A]): A ⇒ Fix[F]
// ana(f: Coalgebra): A ⇒ Fix[F]
val treeFromRules: List[Rule] ⇒ Fix[TreeF] = scheme.ana(build(attrInfoProvider))
val tree: Fix[TreeF]                       = treeFromRules apply rules
```

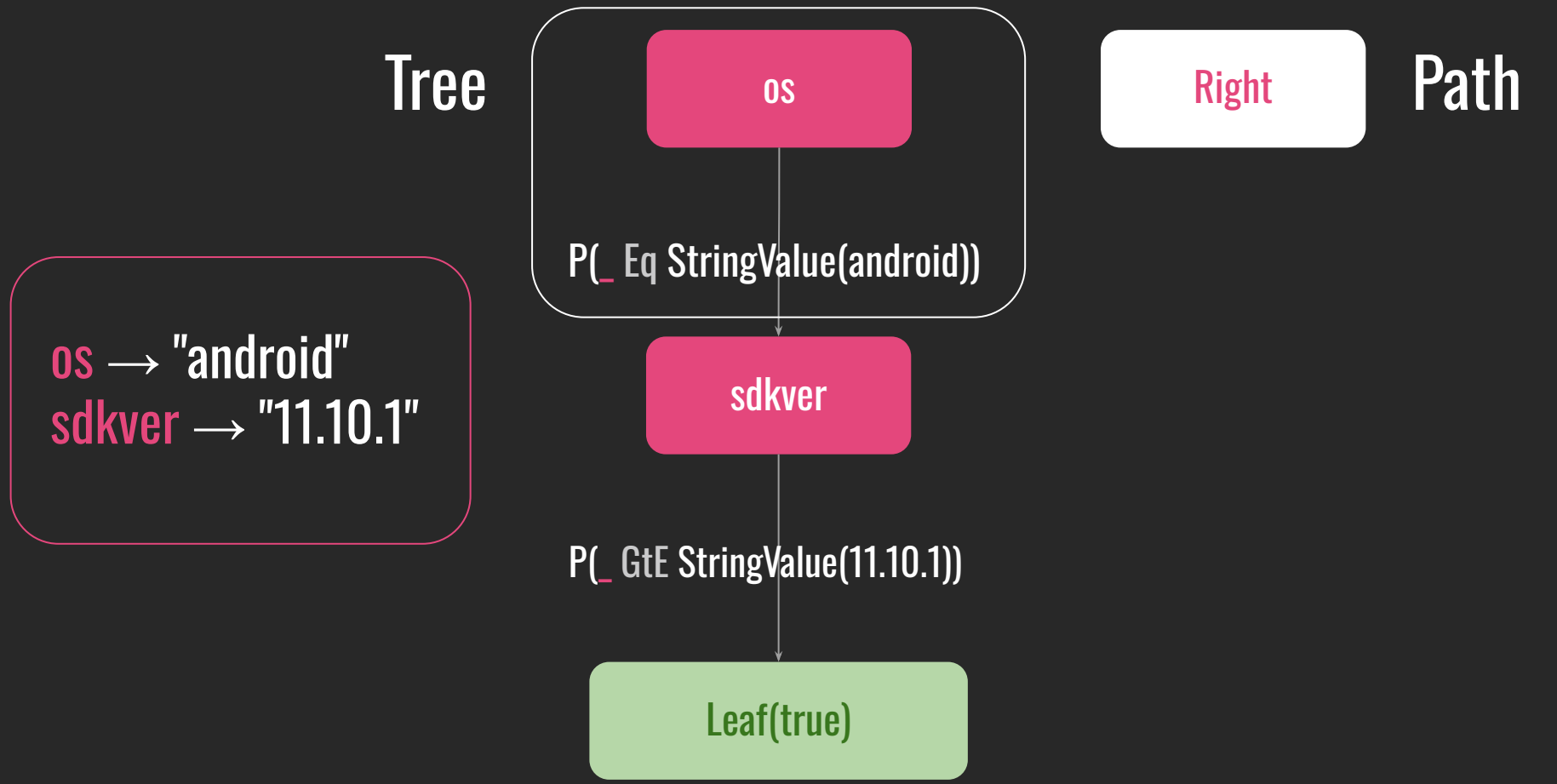
```scala
def fold(getAttrValue: Attr ⇒ Option[Value]): TreeR ⇒ Matching = {
  case Leaf(p)              ⇒ Matching.success(p)
  case Node(attr, children) ⇒
    val step = {
      for {
        matching ← children.findM { case (predicate, _) ⇒ predicate(getAttrValue(attr).getOrElse(NoneValue)) }
        (_, sub) ← matching.toRight(s"No matching predicate found")
      } yield sub
    } leftMap Matching.failure

    step match {
      case Left(matching) ⇒ matching
      case Right(subtree) ⇒ fold(getAttrValue)(subtree)
    }
}
```

```scala
// Fix[TreeF] ⇒ Either[Matching, Fix[TreeF]]]
// Fix[TreeF] ⇒ PathF
private val explore: (Attr ⇒ Option[Value]) ⇒ Coalgebra[PathF, Fix[TreeF]] = attrValues ⇒
  Coalgebra {
    _.unfix match {
      case Leaf(p)              ⇒ Left(Matching.success(p))
      case Node(attr, children) ⇒
        {
          for {
            matching ← children.findM { case (predicate, _) ⇒ predicate(attrValues(attr).getOrElse(NoneValue)) }
            (_, sub) ← matching.toRight(s"No matching predicate found")
          } yield sub
        } leftMap Matching.failure
    }
  }
```

Tree

os → "android"
sdkver → "11.10.1"



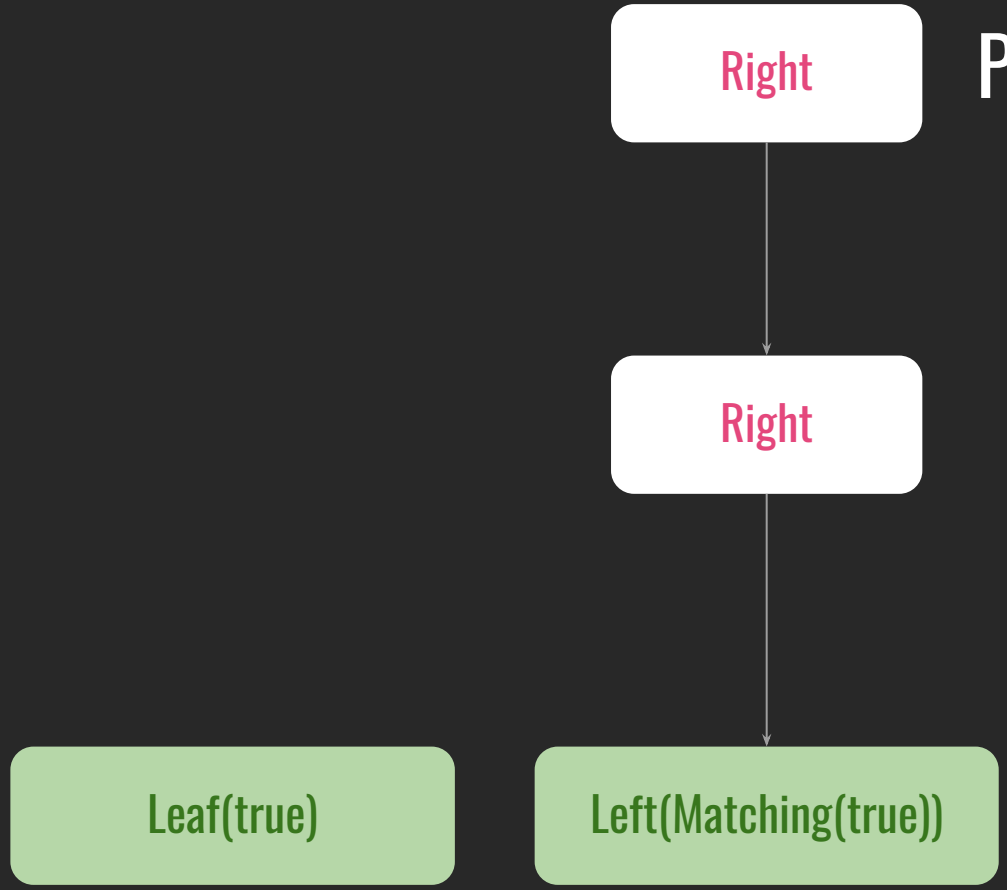

Tree
os
Right
Path
P(_ Eq StringValue(android))
os ⟶ "android"
sdkver ⟶ "11.10.1"
sdkver
P(_ GtE StringValue(11.10.1))
Leaf(true)


Tree
Right
Path
os ⟶ "android"
sdkver ⟶ "11.10.1"
sdkver
P(_ GtE StringValue(11.10.1))
Right
Leaf(true)

Tree

os ⟶ "android"
sdkver ⟶ "11.10.1"

Path

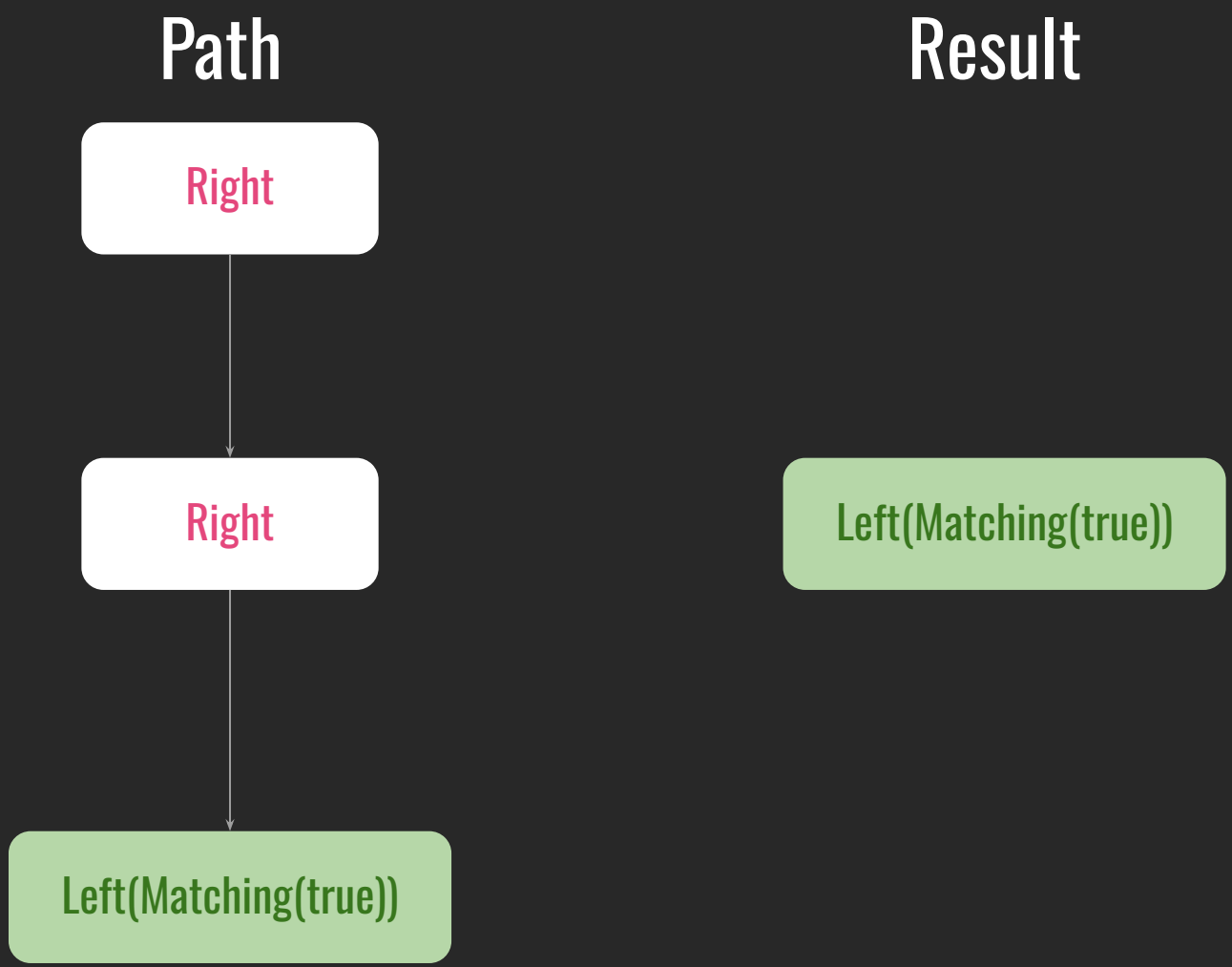


Collapse to a result

```
attrValues ⇒ {
  val pathFromTree: Fix[TreeF] ⇒ Fix[PathF] = scheme.ana(explore(attrValues))
  val path: Fix[PathF]                      = pathFromTree apply tree
```

```
// Either[Matching, A] ⇒ Matching
private val collapse: Algebra[PathF, Matching] = Algebra(_.merge)
```

```
// cata(f: F[A] ⇒ A): Fix[F] ⇒ A
// cata(f: Algebra): Fix[F] ⇒ A
val matchingResultFromPath: Fix[PathF] ⇒ Matching = scheme.cata(collapse)

matchingResultFromPath apply path
```


Path
Result
Right
Right
Left(Matching(true))
Left(Matching(true))

Final implementation

```
def matcher(attrInfoProvider: AttrInfoProvider)(rules: List[Rule]): (Attr ⇒ Option[Value]) ⇒ Matching = {
  // ana(f: A ⇒ F[A]): A ⇒ Fix[F]
  // ana(f: Coalgebra): A ⇒ Fix[F]
  val treeFromRules: List[Rule] ⇒ Fix[TreeF] = scheme.ana(build(attrInfoProvider))
  val tree: Fix[TreeF]                       = treeFromRules apply rules

  attrValues ⇒ {
    val pathFromTree: Fix[TreeF] ⇒ Fix[PathF] = scheme.ana(explore(attrValues))
    val path: Fix[PathF]                      = pathFromTree apply tree

    // cata(f: F[A] ⇒ A): Fix[F] ⇒ A
    // cata(f: Algebra): Fix[F] ⇒ A
    val matchingResultFromPath: Fix[PathF] ⇒ Matching = scheme.cata(collapse)

    matchingResultFromPath apply path
  }
```

Final implementation

```
def matcher(attrInfoProvider: AttrInfoProvider)(rules: List[Rule]): Matcher = {
  val tree = scheme.ana(build(attrInfoProvider)) apply rules

  sample ⇒ scheme.hylo(collapse, explore(sample)) apply tree
}
```

Usage

- Feature flag service (A/B testing)
- Limit traffic sent to the client
- Model complex system behaviour

Benefits

- Scalability
- Flexibility
- Efficiency

Drawbacks

- Complexity
- Debugging
- Validation

Thanks :)