Language features à la carte

Bachelor semester project

Valentin Aebi, IN-Ba6

17.06.2022

Introduction

Goal: a tool to restrict Scala to a chosen subset of its features

- Syntactic analysis
  - Blacklisting
  - Whitelisting
- Semantic analysis → Qifan
- Refactoring imperative code into its functional equivalent

Syntactic analysis

- Scalameta syntax trees

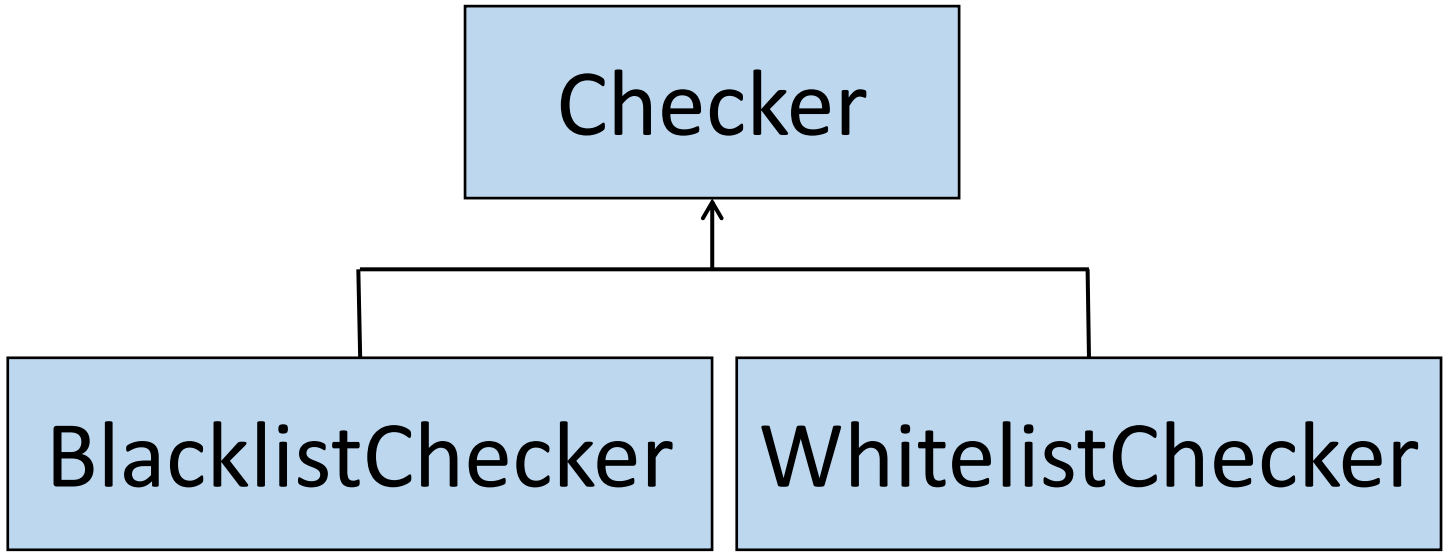
```
Source(List(
  Defn.Def(
    Nil,
    Term.Name("main"),
    Nil,
    List(List(Term.Param(Nil, Term.Name("args"), ..., None))),
    Some(Type.Name("Unit")),
    Term.Block(List(
      Term.Apply(
        Term.Name("println"), List(Lit.String("Hello world!")))
    ))
  )
))
```

The checker trait

```
def checkNode(node: Tree): List[Violation]

final def checkTree(tree: Tree): CheckResult

final def checkCodeString(
        dialect: Dialect,
        sourceCode: String
): CheckResult
```

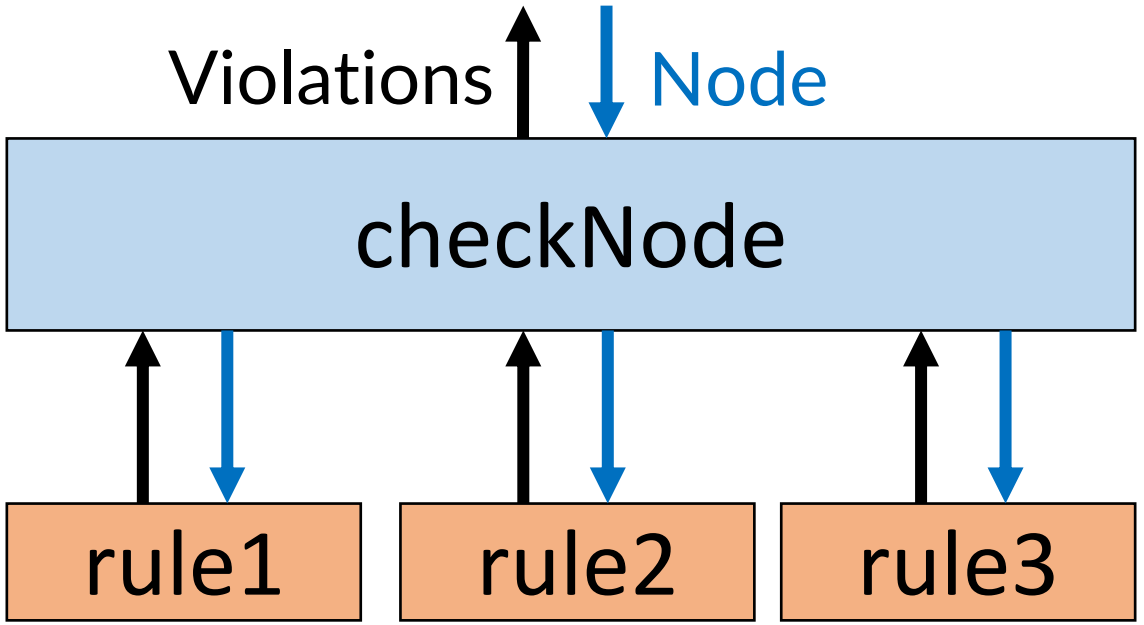


Blacklist checker

```
class BlacklistChecker private(rules: List[BlacklistRule])
                extends Checker {

        override def checkNode(node: Tree): List[Violation]

}
```



Blacklist rules

```
trait BlacklistRule {
  val checkFunc: PartialFunction[Tree, List[Violation]]
}
```

```
case object NoNull extends BlacklistRule {
  override val checkFunc: PartialFunction[Tree, List[Violation]] = {
    case nullKw: Lit.Null =>
      Violation(nullKw, "usage of null is forbidden").toSingletonList
  }
}
```

Result of a check

```
def checkTree(tree: Tree): CheckResult
```

CheckResult ADT:

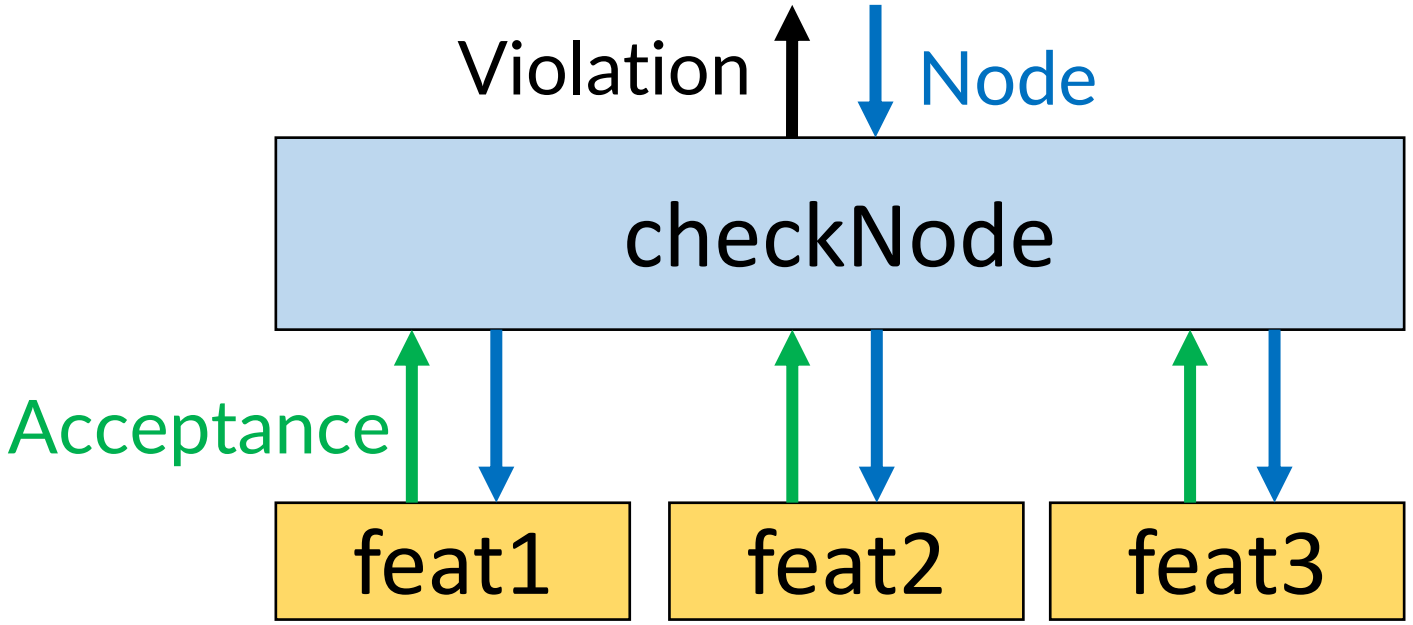
- Valid
- Invalid → list of violations
- ParsingError → exception

Whitelist checker

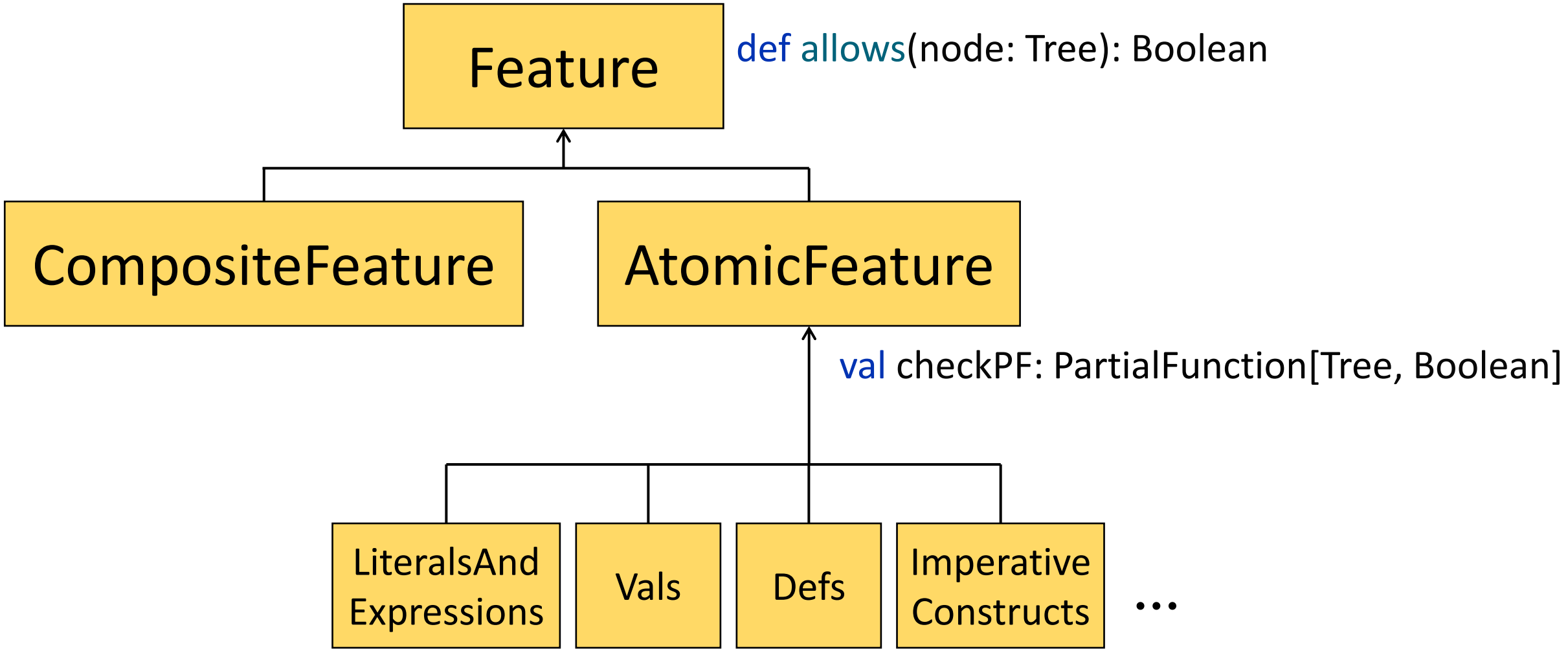
```
class WhitelistChecker private(allowedFeatures: List[Feature])
                extends Checker {

        override def checkNode(node: Tree): List[Violation]

}
```



Features


Feature
def allows(node: Tree): Boolean
CompositeFeature
AtomicFeature
val checkPF: PartialFunction[Tree, Boolean]
LiteralsAnd Expressions
Vals
Defs
Imperative Constructs
...

Features

```
case object ForExpr extends AtomicFeature {
  override val checkPF: PartialFunction[Tree, Boolean] = {
    case _: Term.For => true
    case _: Term.ForYield => true
    case _: Enumerator => true
  }
}
```

Features set computation

Given:

- a list of nodes that should be allowed
- a set of available features

Compute: the minimal set of features s.t. all the nodes are allowed

Features set computation

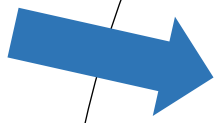
- Computation of all "interesting" subsets of the set of available features

```
1: procedure COMPUTE-MINIMAL-FEATURES-SET(features, AST-nodes)
2:     search-threads ← (∅, AST-nodes)
3:     for f ← features do
4:         new-search-threads ← search-threads
5:         for (selected-features, remaining-nodes) ← search-threads do
6:             if ∃ n ∈ remaining-nodes : n is accepted by f then
7:                 next-rem-nodes ← all n′ ∈ rem-nodes st f does not allow n′
8:                 new-search-threads ← new-search-threads
9:                         ∪ {(selected-features + f, next-rem-nodes)}
10:            end if
11:        end for
12:        search-threads ← new-search-threads
13:    end for
14:    candidates-sets ← ∅
15:    for (selected-features, remaining-nodes) ← search-threads do
16:        if remaining-nodes = ∅ then
17:            candidates-sets ← candidates-sets ∪ {selected-features}
18:        end if
19:    end for
20:    return the set in candidates-sets that has minimal cardinality
21: end procedure
```

Transformation of imperative programs into functional ones

Transformation of imperative programs into functional ones

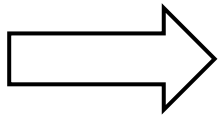
```
def scalarProd(u: List[Int], v: List[Int]): Int = {
  require(u.size == v.size)
  var sum = 0
  val zipped: List[(Int, Int)] = u.zip(v)
  for ((un, vn) <- zipped){
    sum += un * vn
  }
  sum
}
```



```
def scalarProd(u: List[Int], v: List[Int]): Int = {
  require(u.size == v.size)
  def autoGen_0(sum: Int, iterable_0: List[(Int, Int)]): Int = {
    if (iterable_0.nonEmpty) {
      val (un, vn) = iterable_0.head
      autoGen_0(sum + un * vn, iterable_0.tail)
    } else sum
  }
  autoGen_0(0, u.zip(v))
}
```

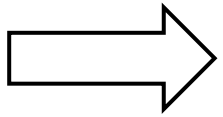
Transformation rules (1)

```
while (cond){
  doSomething()
}
```



```
def method0(): Unit = if (cond){
  doSomething()
  method0()
}
method0()
```

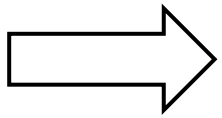
```
do {
  doSomething()
} while(cond)
```



```
def method1(): Unit = {
  doSomething()
  if (cond) method1()
}
method1()
```

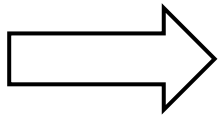
Transformation rules (2)

```
var x = 0
x += 1
```



```
val x = 0
val x_1 = x + 1
```

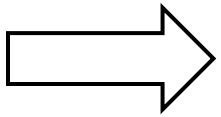
```
var x = 1
while (x < 10){
  x *= 2
}
println(x)
```



```
val x = 1
def method0(x: Int): Int = if (x < 10){
  val x_1 = x*2
  method0(x_1)
} else x
val x_1 = method0(x)
println(x_1)
```

Transformation rules (3)

```
for (x <- ls){
  doSomethingWithX(x)
}
```



```
var iterable_0 = ls
while (iterable_0.nonEmpty){
  val x = iterable_0.head
  doSomethingWithX(x)
  iterable_0 = iterable_0.tail
}
```

Limitations

- Syntactic → Can remove loops and vars, but not mutable objects
- Only a subset of Scala can be transformed
- Does not support variable shadowing
- May require type annotations

Implementation

- RestrictionsEnforcer: checks at the beginning
- NamingContext

```
case class NamingContext(
        currentlyReferencedVars: Map[String, VarInfo],
        currentlyReferencedVals: Map[String, Option[Type]]
)
```

- DisambigIndices: x → x, x_1, x_2, etc.

Inlining (1)

```
def weightedAverage(ls: List[Double], weights: List[Double]): Double = {
  require(ls.size == weights.size)
  val zipped: List[(Double, Double)] = ls.zip(weights)
  var sum = 0.0
  var weightsSum = 0.0
  for ((e, w) <- zipped){
    sum += e*w
    weightsSum += w
  }
  sum / weightsSum
}
```

Inlining (2)

```
def weightedAverage(ls: List[Double], weights: List[Double]): Double = {
  require(ls.size == weights.size)
  val zipped: List[(Double, Double)] = ls.zip(weights)
  val sum = 0.0d
  val weightsSum = 0.0d
  val iterable_0 = zipped
  def autoGen_0(sum: Double, weightsSum: Double, iterable_0: List[(Double, Double)]): (Double, Double) = {
    if (iterable_0.nonEmpty) {
      val (e, w) = iterable_0.head
      val sum_1 = sum + e * w
      val weightsSum_1 = weightsSum + w
      val iterable_0_1 = iterable_0.tail
      autoGen_0(sum_1, weightsSum_1, iterable_0_1)
    } else (sum, weightsSum)
  }
  val (sum_2, weightsSum_2) = autoGen_0(sum, weightsSum, iterable_0)
  sum_2 / weightsSum_2
}
```

Inlining (3)

```
def weightedAverage(ls: List[Double], weights: List[Double]): Double = {
  require(ls.size == weights.size)

  def autoGen_0(sum: Double, weightsSum: Double,
                          iterable_0: List[(Double, Double)]): (Double, Double) = {
    if (iterable_0.nonEmpty) {
      val (e, w) = iterable_0.head
      autoGen_0(sum + e * w, weightsSum + w, iterable_0.tail)
    } else (sum, weightsSum)
  }

  val (sum_2, weightsSum_2) = autoGen_0(0.0d, 0.0d, ls.zip(weights))
  sum_2 / weightsSum_2
}
```

Demo

Thank you for your attention!

Questions?