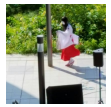
Scalafix ルール作成について

pixiv 株式会社 **Javakky** 

導入

皆さんは Scalafix を知っていますか？

Scalafix は Scala 用の Linter であり自動リファクタリングツールです。

Scalafix のよいところは、

- 自動書き換えができる
- CI からも実行ができる

だと考えています。

CI からも実行ができるので、エディタに依らないコーディングサポートができます。

Scalafix にあるルールは以下のようなものがあり、弊社でもこれらを採用しています。

- インポートの自動削除やソート (https://github.com/liancheng/scalafix-organize-imports)
- 型アノテーションの追記 (ExplicitResultTypes)
- オーバーライド時の () の有無を揃える (https://github.com/ohze/scala-rewrites)

ここまできたら、 Scala のリファクタリング提案をここに集約したい！と考えました。

```
rules = [
  RemoveUnused
  NoAutoTupling
  NoValInForComprehension
  ProcedureSyntax
  fix.scala213.NullaryOverride
  fix.scala213.FinalObject
  fix.scala213.Any2StringAdd
  fix.scala213.Varargs
  fix.scala213.ExplicitNonNullaryApply
  fix.scala213.ExplicitNullaryEtaExpansion
  ExplicitResultTypes
  OrganizeImports
  CheckIsEmpty
  ZeroIndexToHead
  NonCaseException
  UnnecessarySemicolon
]

// 詳細な設定は割愛
```

.scalafix.conf

scalafix-pixiv-rule

そこで scalafix-pixiv-rule の開発を始めました。

このルールが目指すのは、IDEに依存しないコーディング支援が得られる世界です。

ルールは基本的に、 IntelliJ の CodeInspection を参考にしていますが、社内で使われているコーディング規約のためのものもあります。

実装済みのルールをいくつか見ていきます。

`UnnecessarySemicolon` は、行末のセミコロンを削除してくれるルールです。

チームのリポジトリに実行したところ、5ヶ所のセミコロンがかくれんぼしていました。

後の方でルールの作り方について話すので、細かい実装についてはそこで説明します。

`NonCaseException` は、Exception を継承した case class に警告を出してくれるルールです。

これはチームで独自の規約で、エラーの階層化や比較のために採用しています。

`CheckIsEmpty` と `ZeroIndexToHead` はコレクションでより適切なメソッドを利用するように置き換えるもので、 IntelliJ Scala プラグインのアルゴリズムを参考にしています。

現在提供しているルールはここまでですが、これは将来的に実装したいもののほんの一部です。

続報をお待ちください。

scalafix のルール実装

ここからは Scalafix のルールの作り方を話していきます。

Scalafix には Syntactic ルールと Semantic ルールがあります。

Syntactic Rule

Syntactic ルールはその名のとおり構文をチェックするルールで、先ほど紹介した中では `UnnecessarySemicolon` が該当します。

このルールでは、あるシンボルの型を参照するようなルールは作れませんが、構文的なエラーは十分検出することができます。

`SyntacticRule` を継承したクラスを作成し、 `fix` をオーバーライドすることでルールとして利用することができます。

渡された `doc.tokens` には構文木が詰まっていて、 `collect` などで走査しながら変換ルールを追加していくことで実装することができます。

例えば、次のコードは以下のようなトークンに分解されます。

(※実際のコードではありません。疑似コードです)

```
val x = 3
[
  Token$KwVal("val"),
  Token$Space(" "),
  Token$Ident("x"),
  Token$Space(" "),
  Token$Equals("="),
  Token$Space(" "),
  Token$Constant$Int("3")
]
```

それでは `UnnecessarySemicolon` のコードを見てみましょう。

今回は、 `Token.Semicolon` の次に `[Token.CR](<http://Token.CR>)` などの行末があれば `semicolon` トークンを空文字列に置換することができます。

```
class UnnecessarySemicolon extends SyntacticRule("UnnecessarySemicolon") {
  override def fix(implicit doc: SyntacticDocument): Patch = {
    doc.tokens.zipWithIndex.collect { case (semicolon: Token.Semicolon, i) =>
      doc.tokens(i + 1) match {
        case _ @(Token.CR() | Token.LF() | Token.EOF()) =>
          Patch.replaceToken(semicolon, "")
        case _ => Patch.empty
      }
    }.asPatch
  }
}
```

`Patch` には、他にもたとえばこのようなメソッドがあります。

- `empty` : 何も書き換えを行わない
- `replaceToken(token: Token, toReplace: String)` : トークンのあった位置を文字列で置換する
- `addAround(tok: Token, left: String, right: String)` : トークンの左右に文字列を追加する
- `lint(msg: Diagnostic)` : コードには変更を加えずに警告を出力する

Semantic Rule

次に、Semantic ルールを見ていきます。

このルールでは、 Syntactic ルールと比べてより複雑なことを実現することができます。

ここでは、 `doc.token` の代わりに `doc.Tree` を利用します。

では、先ほどのコードをツリーで `collect` に通すとどうなるでしょうか？
(※実際のコードではありません。疑似コードです)

```
val x = 3
[
  Source("val x = 3"),
  Defn$Val("val x = 3"),
  Pat$Var("x"),
  Term$Name("x"),
  Lit$Int("3")
]
```

さらに、 `Defn$Val` は以下のように定義されていて `Val` にマッチしてからも中身を階層的に参照することができます。

```
class Val(
  mods: List[Mod],
  pats: List[Pat] @nonEmpty,
  decltpe: Option[scala.meta.Type],
  rhs: Term
) extends Defn
```

Treeの各具象クラスの定義は、 Tree.scala から見ることができます。

より具体的な例として、 `ZeroIndexToHead` のコードを見ていきましょう。

ここでは、あるシンボルに対して `(0)` を適用することにマッチしている場合に `(0)` に該当する部分を `head` というメソッドの適用に置換しています。

```
class ZeroIndexToHead extends SemanticRule("ZeroIndexToHead") {
  override def fix(implicit doc: SemanticDocument): Patch = {
    doc.tree.collect {
      case t @ Term.Apply(x1, List(Lit.Int(0))) =>
        Patch.replaceTree(
          t,
          Term.Select(x1, Term.Name("head")).toString
        )
    }.asPatch
  }
}
```

しかし、これでは `x1(0)` が本当に `x1.head` に置換しても問題ない確証が持てません。なので、 `x1` が想定される型 (ここでは、 `Seq` かつ `IndexedSeq` ではないクラス) かどうかをチェックしたいと思います。

Symbol から型を取り出す

私のアプローチとしては、 Symbol が表す文字列から完全修飾クラス名を類推するというものです。

[SemanticType を Class_] に変換できるようにする #6

ここで、 Scala の Symbol は以下のように表されます

1. `Package` : 各記号名を `/` で連結する
2. `Class` または `Trait` : パッケージ名 + シンボル名 + `#`
3. `Object` : パッケージ名 + シンボル名 + `.`

まず、圧倒的にパターンとして多いのが 2. のパターンなので、愚直に変換してみます。

```
Class.forName(str.replace('/', '.').init)
```

1. のパターンを考慮するなら末尾の文字が `#` であることを確認すべきです。

しかし、多くの場合 `Object` が渡される場合にはコンパニオンオブジェクトの `apply` 呼び出しの省略形が渡されている場合が多いのではないかと考えたため、暫定的に同じ名前のクラス名を参照するような実装になっています。これは今後対応すべき問題です。

また、 4. のパターンに対応するため、もし上記のコードで変換ができなかった場合にはどこかのオブジェクト (おそらくはパッケージオブジェクト) が持っている `Type` であるのではないかと推論してクラスに変換しています。 (`Predef.String`)

```
val str = str.replace('/', '.').init
val lastDot = str.lastIndexOf('.')
// 最後の `.` までの文字列をオブジェクトに変換する
val objCls = Class.forName(str.take(lastDot) + "$")
import scala.reflect.runtime.universe.{TypeName, runtimeMirror}
val mirror = runtimeMirror(getClass.getClassLoader)
// 前項で生成したオブジェクトから `type` を取り出す
mirror.runtimeClass(mirror.classSymbol(objCls).toType.decl(
  TypeName(str.drop(lastDot + 1))
).typeSignature.dealias.typeSymbol.asClass)
```

ただ、先ほども話した通りこの手法は場当たり的で省略も多いため、よりスマートにシンボルから型を取る方法があれば教えていただきたいです。

抽出子を利用したコード

型を取り出す話はこれぐらいにして、 `CheckIsEmpty` の例を使って抽出子を利用したコーディングのパターンを見ていきましょう。

この項では抽出子と呼ばれる `unapply` メソッドを利用しています。

抽出子はパターンマッチで指定すると呼び出しを行って戻り値を変数として受け取ることができます。言葉で説明するのは難しいので、コードを例示します。 (※コードは一例です)

```
"test@example.com" match {
  case Email(local, domain) => println(s"local: ${local}, domain: ${domain}")
  case  _ => println("Isn't Email!")
}

object Email {
  def unapply(str: String): Option[(String, String)] = {
    val parts = str split "@"
    if (parts.length == 2) Some(parts(0), parts(1)) else None
  }
}
```

では実際に利用しているコードを見てみましょう。

`CheckIsEmpty` 以下が `fix` メソッドのすべてです。ここでは具体的にどういうパターンがヒットするかは記載されていません。

```
class CheckIsEmpty extends SemanticRule("CheckIsEmpty") {
  override def fix(implicit doc: SemanticDocument): Patch = {
    doc.tree.collect {
      // IsDefined かつ書き換えが必要なパターンであれば
      case t @ IsDefined(x1, rewrite) if rewrite && isType(x1, classOf[Option[Any]]) =>
        Patch.replaceTree(t, Term.Select(x1, Term.Name("isDefined")).toString())
      // NonEmpty かつ書き換えが必要なパターンであれば
      case t @ NonEmpty(x1, rewrite) if rewrite && CheckIsEmpty.isTypeHasIsEmpty(x1) =>
        Patch.replaceTree(t, Term.Select(x1, Term.Name("nonEmpty")).toString())
      // IsEmpty かつ書き換えが必要なパターンであれば
      case t @ IsEmpty(x1, rewrite) if rewrite && CheckIsEmpty.isTypeHasIsEmpty(x1) =>
        Patch.replaceTree(t, Term.Select(x1, Term.Name("isEmpty")).toString())
    }.asPatch
  }
}
```

とは言ったもののどこかにはパターンを記載する必要があって、例えば `IsEmpty` オブジェクトの中に定義されています。

```
private object IsEmpty {
  def unapply(tree: Tree)(implicit doc: SemanticDocument): Option[(Term, Boolean)] = {
    tree match {
      // `seq.isEmpty` は変換不要だが IsEmpty ではある
      case _ @Term.Select(x1: Term, _ @Term.Name("isEmpty")) => Some(x1, false)
      // `seq.size == 0`
      case _ @Term.ApplyInfix(
            Term.Select(x1: Term, _ @(Term.Name("size") | Term.Name("length"))),
            Term.Name("=="),
            Nil,
            List(Lit.Int(0))
          ) => Some((x1, true))
      case _ @Term.ApplyUnary(Term.Name("!"), NonEmpty(x1, _)) => Some((x1, true))
      case _ @Term.ApplyUnary(Term.Name("!"), IsDefined(x1, _)) => Some((x1, true))
      // option == None
      case _ @Term.ApplyInfix(x1: Term, _ @Term.Name("=="), Nil, List(Term.Name("None")))
          if isType(x1, classOf[Option[Any]]) => Some((x1, true))
      // None == option
      case _ @Term.ApplyInfix(Term.Name("None"), _ @Term.Name("=="), Nil, List(x1: Term))
          if isType(x1, classOf[Option[Any]]) => Some((x1, true))
      case _ => None
    }
  }
}
```

今後の課題としては、

- クラスの持つ `Type` や再帰的な `Type` の参照があった場合に対応 (？)
  - `java.lang` 系を参照するときだけこれが渡されるのか、別名を利用すると全てこうなるのか調査
- 型変数が返却されるパターンでの `Class[_]` 取得
  - 周囲のシンボルから引いてくる
- Object が返った際に、そのまま `Term.Apply` されていたら `apply` メソッドの方を見に行く
- プロジェクト側で追加された依存関係の `Class[_]` 取得
  - プロジェクトの成果物を突っ込んだクラスローダーの生成 (？)
- 副作用のあるコードの検出

セットアップと利用したツール

ここまで、実際に構文ルールを実現するためのコードについてお話ししてきましたが、ここからは scalafix ルールを立ち上げてから公開するまでの流れについて話していきたいと思います。

Setup · Scalafix

scalafmt

入れるだけです。 `sbt scalafmtAll` してください。

Installation · Scalafmt

scalafix

もちろん scalafix も走らせます。

Installation · Scalafix

`projectMatrix` には実行できないっぽかったので、専用のサブプロジェクトを定義しました。

```
lazy val src = (project in file("rules"))
  .settings(
    libraryDependencies ++= Seq(
      "ch.epfl.scala" %% "scalafix-core" % V.scalafixVersion,
      "org.scalatest" %% "scalatest" % "3.2.11" % "test"
    ),
    scalacOptions ++= Seq(
      "-deprecation",
      "-feature",
      "-Ywarn-unused:imports,locals,patvars"
    ),
    semanticdbEnabled := true,
    semanticdbVersion := scalafixSemanticdb.revision,
    publish / skip := true,
  )
```

`sbt "src/scalafixAll"` で整形できます。(`input`, `output` はあえて規約に外れたコードを書くことが多いので掛けていません)

sbt-license-report

OSSとして公開するので、利用しているライブラリのライセンスも明記したいな〜と思って入れてみました。

https://github.com/sbt/sbt-license-report

以下の設定をして `sbt "src/dumpLicenseReport"` を行うことで NOTICE.md に依存ライブラリとライセンスの一覧が出力できました。

```
lazy val src = (project in file("rules"))
  .settings(
    licenseReportTitle := "NOTICE",
    licenseReportDir := `Repository Name`.base,
    licenseReportTypes := Seq(MarkDown)
  )
```

sbt-ci-release

github のリリースに反応して maven に push してくれるので非常に便利です。

https://github.com/sbt/sbt-ci-release