Scala メタプログラミング

Scala は関数型言語パラダイムを汲んでおりメタプログラミングをサポートしている。 メタプログラミングについて興味があるため、Scala のメタプログラミングでは何が行えるのか調査を行った。 その中で Scala のメタプログラミングの要素としてインラインが挙げられており、それについて調査を行った。

目次

1. 概要 1
2. メタプログラミング 1
  2.1 Scala マクロ 2
3. インライン 2
4. まとめと今後 4
5. 付録 4
Bibliography 4

1. 概要

まず Scala におけるメタプログラミングについて調査を行った。 調査の出発点として Eugene Burmako の [1] を取り扱う。 Scala 2.10 のリリースの際にマクロの機能が導入された。 この論文はその際に書かれたものである。 これに付随して Scala の公式ドキュメントとして同様に Eugene Burmako によって書かれたものが [2] である。 日本語のドキュメントとして Eugene Yokota が翻訳したものが [3] として存在する。 これらとは別に Scala 3 に基づいたドキュメントとして [4], [5] が存在する。

2. メタプログラミング

Eugene Burmako の [1] を基にメタプログラミングについて理解する。

コンパイル時メタプログラミングはコンパイル時にプログラムをアルゴリズムで構築するものだと考えることができる。 プログラマーが自分でプログラムの一部を書くのではなくプログラムの一部を生成できるようにする目的で使われることが多い。 つまりメタプログラミングは他のプログラムについての知識を持ち、それらを操作できるプログラムである。

言語やパラダイムを超えて、この種のメタプログラミングは非常に有用である。 下記の技術はこれによって支えられている。

- 言語仮想化 (language virtualization)
  - 元のプログラミング言語のセマンティクスをオーバーロードまたはオーバライドする
- 外部ドメイン固有言語の埋め込み
  - 外部ドメイン固有言語をホスト言語に対して統合する
- 自己最適化
  - プログラム自身のコードの解析に基づいて最適化を自己適用する

- ボイラーテンプレートの生成
  - 基盤となる言語で容易に抽象化できない反復パターンを自動化する

2.1 Scala マクロ

Scala 2.10 では Scala Macros が導入され、Scala のコンパイル時メタプログラミングを実現した。 これによってコンパイラは Scala プログラム内の特定のメソッドをメタプログラムまたは マクロ*(macros)* として認識し、それらをコンパイルの特定のタイミングで呼び出せる。 呼び出し時にマクロに対してコンパイラの文脈が渡される。 コンパイラの文脈には、現在コンパイルされているプログラムのコンパイラ表現と解析、型チェック、エラーレポートなどのコンパイラ機能を提供する API が含まれている。 文脈の使用可能な API を使用することでマクロはコンパイルに影響を与えることができる。 例えばコンパイルされるコードを変更したり、型推論に影響を与えるなどである。

3. インライン

インライン化は一般的なコンパイル時のメタプログラミング手法であり、通常はパフォーマンスの最適化を達成するために使用される。 Scala 3 では、インライン化の概念がマクロを使用したプログラミングへの入り口を提供する。

下記はインラインの例である。 具体的には `logging` と `log` が `inline` で宣言されている。 `logging` が `inline` で宣言されているため `log` の `if-then-else` 式の評価はランタイム時ではなくコンパイル時に行われる。 参考: https://docs.scala-lang.org/scala3/reference/metaprogramming/inline.html

```
object FactObj:
  object Config:
    inline val logging = false

  object Logger:
    private var indent = -1

    inline def log[T](msg: String, indentMargin: ⇒ Int)(op: ⇒ T): T =
      if Config.logging then
        println(s"${"  " * indent}start $msg")
        indent += indentMargin
        val result = op
        indent -= indentMargin
        println(s"${"  " * indent}$msg = $result")
        result
      else op
  end Logger

  var indentSetting = 2

  def factorial(n: BigInt): BigInt =
    Logger.log(s"factorial($n)", indentSetting) {
      if n == 0 then 1
      else n * factorial(n - 1)
    }


```

```
28    def main(args: Array[String]) =
29      println(factorial(3))
30
31  end FactObj
```

logging の値は false になっているためコンパイル後の log には else 節の op のみが含まれる。 factorial の中身は下記と等価であるといえる。

```
1  def factorial(n: BigInt): BigInt =
2    if n == 0 then 1
3    else n * factorial(n - 1)
```

logging が true の場合は下記になる。

```
1  def factorial(n: BigInt): BigInt =
2    val msg = s"factorial($n)"
3    println(s"${"  " * indent}start $msg")
4    Logger.inline$indent_=(indent.+(indentSetting))
5    val result =
6      if n == 0 then 1
7      else n * factorial(n - 1)
8    Logger.inline$indent_=(indent.-(indentSetting))
9    println(s"${"  " * indent}$msg = $result")
10   result
```

このときの実行結果は下記になる。

```
1  % scala-cli run . --server=false --main-class FactObj
2  start factorial(3)
3    start factorial(2)
4        start factorial(1)
5            start factorial(0)
6            factorial(0) = 1
7        factorial(1) = 1
8    factorial(2) = 2
9  factorial(3) = 6
10 6
```

if 式自体を inline で宣言することができ、その場合は if 式のどのブランチであるかコンパイル時に決定できる必要がある。 同様に match 式も inline で宣言することができる。

ここでは transparent で宣言しており、これによって返す値の型が特定のものに決定できることを指す。 そのため、下記の x は inline で宣言することができる。 transparent の宣言がないと toInt の返却時の型は Int になってしまう。 宣言があると toInt は特定の型を返すことができ、実際に x の型は定数リテラルの 2 である。

```
1  object InlineMatch:
2    trait Nat
3    case object Zero extends Nat
```

```
  case class Succ[N <: Nat](n: N) extends Nat

  transparent inline def toInt(n: Nat): Int =
    inline n match
      case Zero     ⇒ 0
      case Succ(n1) ⇒ toInt(n1) + 1

  def main(args: Array[String]) =
    inline val x = toInt(Succ(Succ(Zero)))
    val x2: Int = x
    val x3: 2 = x

    // error because x is literal constant type
    // val x4: 3 = x

end InlineMatch
```

つまり transparent 宣言を用いることによってこの場合 toInt の返却する型は Int ではなく Int に属する型リテラルであり、toInt の返却する型を透過的に変更している。型チェックはランタイム時ではなくコンパイル時に実行されるため、x3 のような表現が可能になる。 実際に x4 はコンパイル時にエラーになることに加え、コーディング時に Laguage Server を通して静的に検出される。

4. まとめと今後

Scala のメタプログラミングの論文に触れることで、メタプログラミングそのものへの理解と Scala がメタプログラミングの実装、検証として利用されていることが分かった。 実際に Scala のメタプログラミングの要素であるインラインに触れた。 インラインを用いることでランタイム時ではなくコンパイル時にあらかじめ計算することで最適化を行うことができる。 加えて transparent を用いることで抽象型で宣言された関数でも透過的に特定の型を返すことができる。

本来は Scala のマクロまで触れたかったが、時間が足らなかったため今後の課題とする。 少し具体的に調査した内容について述べると、マクロを用いることで正しく型づけされた状態で多段階計算を実現することができる。 日本語に記事として [6] が詳細に解説していた。

5. 付録

本レポート上で書かれたコードは https://github.com/blank71/scala-learn しているため、そこから利用することができる。 同様に本レポートの原本もアップロードしてある。

Bibliography

[1] E. Burmako, “Scala macros: let our powers combine! on how rich syntax and static types work with metaprogramming,” in *Proceedings of the 4th Workshop on Scala*, in SCALA '13. Montpellier, France: Association for Computing Machinery, 2013. doi: 10.1145/2489837.2489840.

[2] E. Burmako, “Macros.” Accessed: Jan. 18, 2025. [Online]. Available: https://docs.scala-lang.org/overviews/macros/usecases.html

[3] E. Burmako and E. Yokota, “Macros.” Accessed: Jan. 18, 2025. [Online]. Available: https://docs.scala-lang.org/ja/overviews/macros/usecases.html

[4] “Macros in Scala 3.” Accessed: Jan. 18, 2025. [Online]. Available: https://docs.scala-lang.org/scala3/guides/macros/

[5] “Metaprogramming.” Accessed: Jan. 18, 2025. [Online]. Available: https://docs.scala-lang.org/scala3/reference/metaprogramming/index.html

[6] “あらゆるプログラミング言語の最先端を行く Scala 3 のマクロ.” [Online]. Available: https://tarao.hatenablog.com/entry/scala3-multi-stage