Лабораторная работа № 4 «Case-классы и сопоставление с образцом в Scala»

3 апреля 2024 г.

Илья Афанасьев, ИУ9-61Б

Цель работы

Целью данной работы является приобретение навыков разработки case-классов на языке Scala для представления абстрактных синтаксических деревьев.

Индивидуальный вариант

Абстрактный синтаксис выражений с let:

`Expr → Expr + Expr | Expr * Expr | VARNAME | let VARNAME = Expr in Expr`

Требуется написать функцию `letsOptimize : Expr => Expr`, которая выполняет оптимизацию выражения:

- если в выражении есть общие подвыражения, то общий код выносится в let: `(x + y) * (x + y) → let v0 = x + y in v0 * v0`;
- если переменная, определённая let'ом, в выражении встречается только один раз, следует удалить let и подставить вместо переменной выражение: `letx = y + z in a * x → a * (y + z)`.

Указание. Считайте, что в выражении в операциях let имена переменных не могут повторяться (например, `(let x = y + z in x + y) * (let x = z * z in y * z)` — ошибочное выражение) и после let нельзя использовать свободную переменную выражения (например, `x + (let x = y * y in x * y)` — ошибочное выражение). Эти ограничения существенно упростят преобразование выражений.

Реализация

```
enum BinaryOp:
  case Plus, Star

abstract class Expr
```

```
case class VarExpr(name: String) extends Expr
case class BinaryExpr(op: BinaryOp, lhs: Expr, rhs: Expr) extends Expr
case class LetExpr(v: VarExpr, subst: Expr, to: Expr) extends Expr

object VarExpr {
  private val base = "v";
  private var counter = -1;

  def nextName(): String = {
    counter += 1
    base + counter
  }
}

object Main {
  def occurenciesOf(what: VarExpr, where: Expr): Int = where match {
      case `what` => 1
      case BinaryExpr(_, lhs, rhs) => {
        occurenciesOf(what, lhs) + occurenciesOf(what, rhs)
      }
      case LetExpr(_, subst, to) => {
        occurenciesOf(what, subst) + occurenciesOf(what, to)
      }
      case other => 0
    }

  def replace(what: VarExpr, by: Expr, where: Expr): Expr = where match {
      case `what` => by
      case BinaryExpr(op, lhs, rhs) => {
        BinaryExpr(op, replace(what, by, lhs), replace(what, by, rhs))
      }
      case LetExpr(v, subst, to) => {
        LetExpr(v, replace(what, by, subst), replace(what, by, to))
      }
      case other => other
    }

  def letsOptimize(what: Expr): Expr = what match {
      case BinaryExpr(op, lhs @ BinaryExpr(_, _, _), rhs) if lhs == rhs => {
        val v = VarExpr(VarExpr.nextName())
        LetExpr(v, letsOptimize(lhs), BinaryExpr(op, v, v))
      }
      case BinaryExpr(op, lhs, rhs) => {
        BinaryExpr(op, letsOptimize(lhs), letsOptimize(rhs))
      }
      case LetExpr(v, subst, to) if occurenciesOf(v, to) == 1 => {
```

```
        letsOptimize(replace(v, subst, to))
      }
      case LetExpr(v, subst, to) => {
        LetExpr(v, letsOptimize(subst), letsOptimize(to))
      }
      case other => other
    }

  def main(args: Array[String]): Unit = {
    // x + x → x + x
    val e1 = BinaryExpr(BinaryOp.Plus, VarExpr("x"), VarExpr("x"))
    println(letsOptimize(e1))

    // (x + y) * (x + y) → let v0 = x + y in v0 * v0
    val e2 = BinaryExpr(BinaryOp.Plus, VarExpr("x"), VarExpr("y"))
    val e3 = BinaryExpr(BinaryOp.Star, e2, e2)
    println(letsOptimize(e3))

    // ((x + y) * (x + y) + (x + y) * (x + y)) * z →
    // (let v0 = (x + y) * (x + y) in v0 + v0) * z →
    // (let v0 = (let v1 = x + y in v1 * v1) in v0 + v0) * z
    val e4 = BinaryExpr(BinaryOp.Plus, e3, e3)
    val e5 = BinaryExpr(BinaryOp.Star, e4, VarExpr("z"))
    println(letsOptimize(e5))

    // let x = y + z in a * x → a * (y + z)
    val e6 = BinaryExpr(BinaryOp.Plus, VarExpr("y"), VarExpr("z"))
    val e7 = BinaryExpr(BinaryOp.Star, VarExpr("a"), VarExpr("x"))
    val e8 = LetExpr(VarExpr("x"), e6, e7)
    println(letsOptimize(e8))

    // let y = (c + d) * (c + d) in (let x = y + z in a * x) →
    // let x = (c + d) * (c + d) + z in a * x →
    // a * ((c + d) * (c + d) + z) →
    // a * ((let v0 = c + d in v0 * vo) + z)
    val e9 = BinaryExpr(BinaryOp.Plus, VarExpr("c"), VarExpr("d"))
    val e10 = BinaryExpr(BinaryOp.Star, e9, e9)
    val e11 = LetExpr(VarExpr("y"), e10, e8)
    println(letsOptimize(e11))
  }
}
```

Тестирование

Результат преобразования выражений, написанных в функции `main`:

```
BinaryExpr(Plus,VarExpr(x),VarExpr(x))
LetExpr(
    VarExpr(v0),
    BinaryExpr(Plus,VarExpr(x),VarExpr(y)),
    BinaryExpr(Star,VarExpr(v0),VarExpr(v0))
)
BinaryExpr(
    Star,
    LetExpr(
        VarExpr(v1),
        LetExpr(
            VarExpr(v2),
            BinaryExpr(Plus,VarExpr(x),VarExpr(y)),
            BinaryExpr(Star,VarExpr(v2),VarExpr(v2))
        ),
        BinaryExpr(Plus,VarExpr(v1),VarExpr(v1))
    ),
    VarExpr(z)
)
BinaryExpr(Star,VarExpr(a),BinaryExpr(Plus,VarExpr(y),VarExpr(z)))
BinaryExpr(
    Star,
    VarExpr(a),
    BinaryExpr(
        Plus,
        LetExpr(
            VarExpr(v3),
            BinaryExpr(Plus,VarExpr(c),VarExpr(d)),
            BinaryExpr(Star,VarExpr(v3),VarExpr(v3))
        ),
        VarExpr(z)
    )
)
```

Вывод

В результате выполнения лабораторной работы я приобрёл навыки разработки case-классов на языке Scala для представления абстрактных синтаксических деревьев.