Teoria Współbieżności

Sprawozdanie do zadania domowego

Zadanie V

Stanisław Denkowski

Nr indeksu 305288

8 listopada 2020

Cel zadania

Analizujemy alfabet *A*, gdzie każda litera oznacza akcję, zaś słowo *w* sekwencję akcji. Dla określonego zestawu transakcji na zmiennych należy napisać program w języku funkcyjnym, który określi:

- Relacje zależności i niezależności
- Postać normalną Foaty FNF([w]) śladu [w]
- Graf zależności w postaci minimalnej
- Ponownie wyznaczy postać normalną Foaty - na podstawie grafu

Ułatwieniem zadania jest założenie, że każda transakcja dotyczy dokładnie dwóch zmiennych (po prawej stronie równania, po lewej zawsze jedna zmienna).

Sposób realizacji

Opis programu

Zdecydowałem się na wykorzystanie języka Scala, który daje możliwość programowania funkcyjnego. Poniżej podaję kolejne fragmenty mojego programu - tj. jego kod źródłowy z komentarzem.

Parsowanie danych wejściowych

```
package traces

class Data(val name: String) {
 // Wczytujemy wszystkie linie z pliku o danej nazwie
 val lines: List[String] = readFile(name)

 // Wyznaczamy zadany alfabet
 // spośród wczytanych linii wyszukuje zaczynającą się od "A", i pomijam wszystkie znaki spoza a-z
 lazy val Alphabet: List[Char] = {
   lines.filter(_.startsWith("A")).mkString("").replaceAll("[^a-z]", "").toList
 }

 // Wyznaczamy zadane słowo
 // spośród wczytanych linii wyszukuje zaczynającą się od "w", i pomijam wszystko do "=" włącznie
 lazy val Word: String = {
   lines.filter(_.startsWith("w")).mkString("").dropWhile(_ != '=').tail.trim()
 }

 // Wyznaczamy produkcje
 // spośród wczytanych linii wyszukuje zaczynające się od "(", i pomijam wszystkie znaki spoza a-z
 // otrzymuje krótkie (litera z alfabetu, zmienna przed "=", zmienna po +, druga zmienna po +) - zakładam
poprawność
 lazy val Prods: List[(Char, Char, Char, Char)] ={
   lines.filter(_.startsWith("(")).map(
     line => {
       val simp = line.replaceAll("[^a-z]", "")
       (simp(0), simp(1), simp(2), simp(3))
     }
   )
 }
```

```
 // Funkcja wczytująca wszystkie linie z pliku, w sposób bezpieczny
 def readFile(name: String): List[String] = try{
   val file = scala.io.Source.fromFile(name)
   val lines = try {
     file.getLines.toList
   }
   finally {
     file.close
   }
   lines
 } catch {
   case _: java.io.FileNotFoundException => println("File not found " + name); List()
   case ex: Throwable => println("Exception " + ex.getMessage); List()
 }
}
```

Tworzenie zbiorów zależności - D i niezależności - I

```
package traces

class Productions(val prods: List[(Char,Char,Char,Char)]) {
 // Wyznaczamy zbiór D (zaleznosci), wykorzystując listę prod z obiektu Data
 // Tworzę produkt kartezjański i wybieram tylko te pary, których elementy są w relacji
 lazy val D: Set[(Char, Char)] = {
   prods.flatMap(prodA => {
     prods.map(prodB => (prodA, prodB))
   }).filter(rel => ArelB(rel._1, rel._2)).map(rel => {
     (rel._1._1, rel._2._1)
   }).toSet
 }

 // Wyznaczamy zbiór I (niezależności), wykorzystując listę prod z obiektu Data
 // tworzę produkt kartezjański i wybieram tylko te pary, których elementy nie są w relacji
 lazy val I: Set[(Char, Char)] = {
   prods.flatMap(prodA => {
     prods.map(prodB => (prodA, prodB))
   }).filterNot(rel => ArelB(rel._1, rel._2)).map(rel => {
     (rel._1._1, rel._2._1)
   }).toSet
 }

 // Sprawdzam czy dwa elementy z prods z obiektu Data, są w relacji
 def ArelB(A: (Char, Char, Char, Char), B: (Char, Char, Char, Char)):Boolean ={
   A._2 == B._3 || A._2 == B._4 || B._2 == A._3 || B._2 == A._4
 }
```

Funkcja main programu

```
package traces

import scala.annotation.tailrec

object General {

 def main(args: Array[String]): Unit = {
   // Wczytaj argumenty, a jeśli ich nie ma, to ustaw nazwę na domyślną
   val name = if (args.length==1){
       println(args(0))
       args(0)
   }
   else{
       "in.txt"
   }
   // Wczytujemy i przygotowujemy dane
   val data = new Data(name)
   // Wypisujemy wczytane i przerobione dane
   //  println(data.Alphabet)
   //  println(data.Word)
   //  println(data.Prods)
   // Przygotowujemy produkcje, wykorzystujące wczytane dane
   val prods = new Productions(data.Prods)


   // Wypisujemy wyniki
   print("\nD={")
   prods.D.init.foreach(pair => print(s"$pair,"))
   print(prods.D.last)
   print("}\n\nI={")
   prods.I.init.foreach(pair => print(s"$pair,"))
   print(prods.I.last)
   print("}\n\nFNF([w])=")
   solveFnF(data, prods).foreach(cla => {
       print('(')
       cla.sorted.foreach(print(_))
       print(')')
   })

   val gra =  graph(data, prods)
   println("\n\ndigraph g{")
   gra.foreach(edge => {
       println(s"${edge._1+1} -> ${edge._2+1}")
   })
   data.Word.zipWithIndex.foreach(vertex => {
       println(s"${vertex._2+1}[label=${vertex._1}]")
   })
   print("}\n\nGraphFNF([w])=")
   solveGraphFnF(data, gra).foreach(cla => {
       print('(')
       cla.sorted.foreach(print(_))
       print(')')
   })
   println()
 }
```

Wyznaczanie postaci normalnej Foaty

```
 // Wyznaczamy postać normalna Foaty
 def solveFnF(data: Data, prods: Productions): List[List[Char]] = {
   // countClass tworzy nam tupla literki i numeru klasy do jakiego została przypisana
   // jako argumenty dostaje przerobiona czesc słowa i część słowa do przerobienia
   // (w danym kroku sprawdzamy todo.head)
   // wołamy rekurencyjnie na samym koncu, wiec mozemy zrobić optymalizację rekurencji ogonowej
   // danej literce przypisujemy klase będąca najwieksza z już przerobionych,
   // z którymi literka jest w relacji + 1
   @tailrec
   def countClass(done: List[(Char, Int)] = Nil, todo: List[Char] = data.Word.toList): List[(Char, Int)] = {
       if(todo.isEmpty) done
       else if(!done.exists(rel => prods.D contains(rel._1, todo.head))) countClass((todo.head, 0) :: done,
todo.tail)
       else countClass((todo.head, done.filter(rel => prods.D contains (rel._1,
todo.head)).maxBy(_._2)._2+1) :: done, todo.tail)
   }
   // grupujemy wynik po klasie i przygotowujemy do wypisania
   countClass().groupBy(_._2).toList.sortBy(_._1).map(_._2.map(_._1))
 }
```

Wyznaczenie grafu zależności w postaci minimalnej

```
 // Wyznaczamy graf zależności w postaci minimalnej dla danego słowa
 def graph(data: Data, prods: Productions): List[(Int, Int)] = {
   // Tworzymy graf z zaznaczonymi wszystkimi zależnościami, ale taki by żadna krawędź nie szla
   // od późniejszej litery do wcześniejszej
   // Graf reprezentujemy jako listę krawędzi
   val word: List[(Char, Int)] = data.Word.zipWithIndex.toList
   val compl: List[(Int, Int)] = word.flatMap(pair => {
       word.filter(rel=>{rel._2>pair._2 && prods.D.contains((pair._1, rel._1))}).map(rel => (pair._2,
rel._2))
   })

   // indPath przyjmuje graf w postaci listy krawędzi, krawędź która sprawdzamy czy jest zbędna
   // i obecny wierzchołek
   // podobnie do dfs'a przeszukujemy graf zaczynając w lewym końcu krawędzi i sprawdzamy czy jestesmy
   // w stanie dojść do prawego końca, pomijając bezpośrednie połączenie
   def indPath(graf: List[(Int, Int)], redund: (Int, Int), curr: Int): Boolean = {
       if(curr == redund._2) true
       else graf.filterNot(_ == redund).filter(_._1 == curr).exists(edge => indPath(graf, redund, edge._2))
   }

   // Przygotowujemy do wypisania
   compl.filterNot(edge => indPath(compl, edge, edge._1))
 }
```

Postać normalna Foaty na podstawie zbudowanego grafu zależności

```
 // Wyznaczamy postać normalną Foaty, wykorzystujac wyznaczony wcześniej graf (w postaci listy krawędzi)
 // Algorytm działa bardzo podobnie do solveFnF, tylko zamiast sprawdzania czy literki są w relacji,
 // sprawdza czy istnieje taka krawędź
 def solveGraphFnF(data: Data, gra: List[(Int, Int)]): List[List[Char]] = {
   // Bardzo analogiczna funkcja do countClass z solveFnF
   // Dla każdej krawędzi przypisujemy największy numer, z już wcześniej przypisanych krawędziom,
   // będących poprzednikami mojej krawedzi, wartosci
   @tailrec
   def orderEdges(done: List[((Int, Int), Int)] = Nil, todo: List[(Int, Int)] = gra): List[((Int, Int), Int)]
=
   {
       if(todo.isEmpty) done
       else if(!done.exists(_._1._2 == todo.head._1)) orderEdges((todo.head, 0) :: done, todo.tail)
       else orderEdges((todo.head, done.filter(_._1._2 == todo.head._1).map(_._2).max+1) :: done, todo.tail)
   }

   // Grupujemy i przygotowujemy do wypisania
   val res: List[((Int, Int), Int)] = orderEdges().reverse
   data.Word.zipWithIndex.map(pair => {
       if(!res.exists(_._1._2 == pair._2)) (pair._1, 0)
       else (pair._1, res.filter(_._1._2 == pair._2).map(_._2).max+1)
   }).groupBy(_._2).toList.map(_._2.toList.map(_._1))
 }
}
```

Wynik działania dla przykładowych danych

Przykład 1

Dane wejściowe

(a) x := x + y
(b) y := y + 2z
(c) x := 3x + z
(d) z := y - z
A = {a,b,c,d}
w = baadcb

Wyniki działania programu

D={(b,b),(a,b),(b,d),(c,c),(a,a),(b,a),(c,d),(c,a),(d,b),(a,c),(d,d),(d,c)}

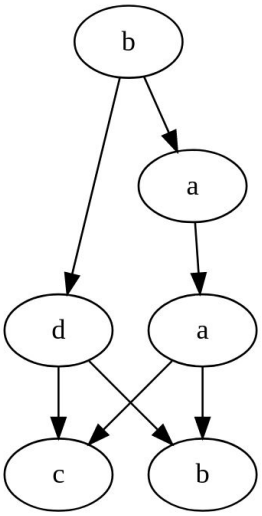
I={(a,d),(b,c),(c,b),(d,a)}

FNF([w])=(b)(ad)(a)(bc)

digraph g{
1 -> 2
1 -> 4
2 -> 3
3 -> 5

3 -> 6
4 -> 5
4 -> 6
1[label=b]
2[label=a]
3[label=a]
4[label=d]
5[label=c]
6[label=b]
}

GraphFNF([w])=(b)(ad)(a)(bc)



Przykład 2

Dane wejściowe

(a) x := x + x
(b) y := y + 2z
(c) x := 3x + z
(d) w := w + v
(e) z := y − z
(f) v = x + v
A = {a,b,c,d,e,f}
w = acdcfbbe

Wyniki działania programu

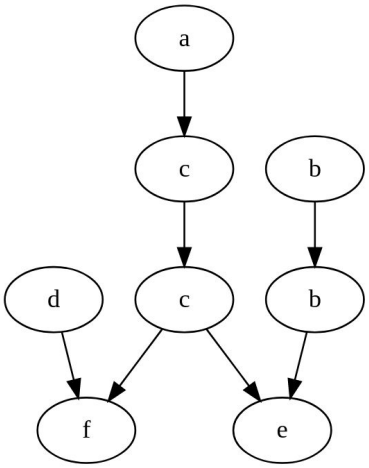
D={(b,b),(d,f),(c,e),(c,c),(f,c),(a,f),(a,c),(d,d),(e,e),(f,d),(e,c),(a,a),(f,a),(c,f),(b,e),(c,a),(f,f),(e,b)}

I={(a,d),(e,f),(a,e),(e,a),(b,a),(c,d),(c,b),(b,c),(d,b),(b,f),(d,c),(f,b),(a,b),(b,d),(d,e),(e,d),(f,e),(d,a)}

FNF([w])=(abd)(bc)(c)(ef)

digraph g{
1 -> 2
2 -> 4
3 -> 5
4 -> 5
4 -> 8
6 -> 7
7 -> 8
1[label=a]
2[label=c]
3[label=d]
4[label=c]
5[label=f]
6[label=b]
7[label=b]
8[label=e]
}

GraphFNF([w])=(abd)(bc)(c)(ef)



Uruchomienie

Program uruchamiamy w następujący sposób, wypisuje rozwiązania na standardowe wyjście:

- sbt build
- sbt run - uruchamia program i jako nazwa pliku do wejścia przyjęty jest “in.txt” (ew. Można w kodzie zmienić)
- sbt run nazwa - uruchamia program, przyjmując plik nazwa, jako dane wejściowe.