Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA

Analizzatore di Liveness in Java

Relazione progetto Analisi dei Sistemi Informatici

Candidati:

Riccardo Astolfi
Matricola VR439692

Giacomo Ferro
Matricola VR439370

Anno Accademico 2018/2019

Indice

1 Introduzione

Abbiamo deciso di realizzare tale progetto per cercare di semplificare la creazione di analizzatori statici di proprietà su linguaggi (nel nostro caso abbiamo implementato un analizzatore di Liveness). Prima di passare a presentare le caratteristiche di tale analizzatore, partiamo fornendo un piccolo ripasso di concetti teorici alla base dell'analisi dei sistemi informatici.
Tali argomenti sono:

- Analisi statica e verifica formale del software
- Caratteristiche del linguaggio in analisi
- Caratteristiche di analisi di Liveness

Dopo aver ripreso queste nozioni di background passeremo a descrivere le caratteristiche del programma descrivendo le sue componenti.
Per sviluppare tale analizzatore abbiamo creato dapprima un parser in JavaCC in ambiente Eclipse e poi ci siamo appoggiati ad una libreria chiamata JGraphT che ci fornisce varie implementazioni di grafi in Java. Abbiamo usato tale struttura dati per creare il CFG dal codice parsato opportunamente. Successivamente, abbiamo creato altre classi in Java che, preso il CFG dal codice, eseguono e ritornano in uscita l'analisi di Liveness del codice. Infine abbiamo utilizzato JavaSwing per creare l'interfaccia grafica di comunicazione con l'user la quale chiede di scrivere il codice da analizzare ed il numero di iterazioni di Liveness da eseguire.

Ai seguenti link sono presenti tutorials su come utilizzare ed eventualmente installare JavaCC, JgraphT, Eclipse IDE Java e JavaSwing sui propri PC:

- https://jgrapht.org/guide/UserOverview
- https://javacc.org/tutorials/lookahead
- https://www.eclipse.org/downloads/packages/
- https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html

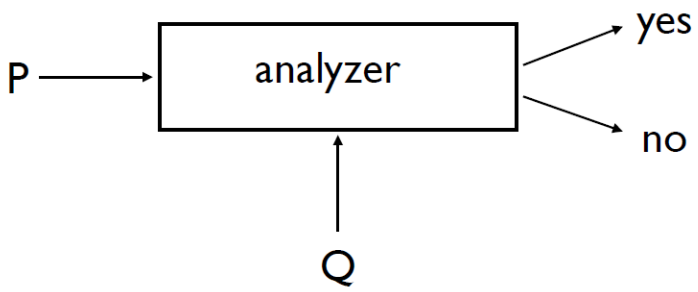
2 Prima parte: Background

2.1 Analisi statica: Overview

L'analisi statica si inserisce nel contesto della verifica formale di software. Normalmente un software in fase di sviluppo non è mai completamente corretto in quanto presenterà caratteristiche che lo renderanno non sicuro. La parte di testing e analisi sono quindi importanti per il controllo qualità e per evitare attacchi informatici. Tuttavia, fare questi controlli costa molto e quindi se il cliente non ha dati sensibili nel software allora in questi casi si può preferire correre il rischio e non investire risorse nella protezione dei dati. Normalmente in un processo di analisi formale occorre trovare un proprietà del software da verificare (ad esempio la liveness) e poi procedere alla verifica di tale proprietà nel sistema. A tal proposito, è importante distinguere la verifica dalla validazione poichè quest'ultima è effettuata generalmente da un essere umano e quindi soggetta maggiormente a soggettività. Durante un processo di analisi formale, potrebbe essere utile passare anche ad un'analisi dinamica. In entrambi i casi (contesto statico e dinamico) abbiamo sempre indecidibilità poichè il primo approccio non sarà completamente esaustivo mentre il secondo non gestirà tutte le risposte per tutti i possibili input.

Le proprietà del software sono innumerevoli e possono essere catalogate in 3 classi fondamentali:

- Proprietà esterne: alcune di queste sono solo verificabili mentre altre solo validabili. A questo proposito, il tempo di esecuzione è la classica proprietà che è facilmente verificabile mentre l'usabilità di un software è validabile poichè esiste una componente soggettiva nella valutazione;
- Proprietà interne: riguardano proprietà interne al sistema quali manuntenibilità, modularità e usabilità;
- Proprietà di affidabilità: sono in generale proprietà molto importanti per il sistema e riguardano generalmente proprietà di sicurezza come correttezza, affidabilità e robustezza;



In figura è mostrata una stilizzazione del processo di decisione per decidere se un sistema verifica o meno una certa proprietà.
Il nostro obbiettivo è quindi quello di dire se una proprietà astratta Q è valida o meno. In questo modo, l'analizzatore statico serve per cercare di prevedere il comportamento del programma P a livello di esecuzione. Nel caso di analisi di liveness occorre partire dalla definizione di un linguaggio e di una semantica per la creazione di un control flow graph.

Prima di passare ad elencare le caratteristiche del linguaggio, mostriamo le caratteristiche fondamentali dell'analisi statica basata su CFG. Tale control flow graph servirà poi per esprimere la semantica operazionale del linguaggio. Partiamo col precisare che tali grafi astraggono la forma di un programma scritto in un certo linguaggio permettendo varie tipologie di analisi locale.
Nell'analizzare il codice tramite CFG abbiamo 3 livelli di analisi:

- Analisi locali: sono tipologie di analisi che riguardano un blocco di istruzioni. Sono quasi sempre convergenti;
- Analisi intra-procedurali: consideriamo un CFG guardando solo l'informazione propagata in questo senza considerare le chiamate esterne ad altri grafi;
- Analisi inter-procedurali: Ogni procedura è rappresentata da un grafo. L'intero programma è rappresentato da una "foresta di grafi";

Nell'analisi tramite grafi si studia come il flusso dei dati è modificato all'interno di un blocco di istruzioni. A questo proposito abbiamo 3 passi che dobbiamo seguire per procedere nell'analisi di un blocco:

- Informazione entrante: è rappresentata dalla combinazione di tutto ciò che esce dai blocchi esterni;
- Informazione uscente: quella che viene generata dal blocco o che sopravvive all'elaborazione. Per calcolarla devo definire quali sono le istruzioni costruttive e distruttive;

Introducendo poi il concetto di dominanza tra nodi, un nodo n è dominato da un certo nodo m se e solo se m sta in tutti i cammini possibili per n. Nell'immagine seguente viene riassunto il concetto di dominanza in cui si raggruppano tutti i possibili dominatori di un nodo (dominatori di dominatori).

$$\textsc{Dom}(n) \stackrel{\text{def}}{=} \{n\} \cup \left(\bigcap_{m \in \textsc{Pred}(n)} \textsc{Dom}(m) \right)$$

2.2 Definizione di linguaggio e CFG

Per procedere con l'analisi di liveness occorre prima definire su che linguaggio deve operare e quali sono i suoi costrutti fondamentali. La semantica operazionale del linguaggio viene definita sempre tramite control flow graph. Il linguaggio su cui operiamo analisi di liveness è definito nel seguente modo:

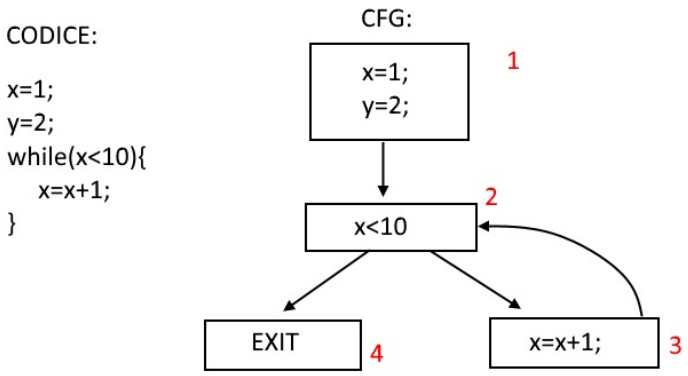
variabili:	x
espressioni:	e
assegnamenti:	x=e;
costrutti condizionali:	if(x op e){ ... }
costrutti iterativi (1):	while(x op e){...}
costrutti iterativi (2):	for(x=e; x op e; x=x+1;){...}

Precisiamo che le guardie saranno solo condizioni semplici e non composte. Nel linguaggio si possono eseguire invece assegnamenti di ogni tipo che coinvolgano variabili ed espressioni aritmetiche. Gli operatori booleani possibili (indicati come *op*) sono: $<$, $>$, $<=$, $>=$, $==$. Infine non è possibile scrivere costrutti annidati nel linguaggio. In altre parole, i corpi di costrutti condizionali ed iterativi sono formati solo da uno o più assegnamenti.
Viene definita per ogni istruzione la trasformazione di stato che comporta. I nodi del CFG sono i punti di programma mentre gli archi sono i passi computazionali etichettati con la corrispondente azione del programma. In particolare, i nodi del grafo sono i cosiddetti basic blocks ovvero una sequenza massimale di istruzioni senza branch interni e con un singolo entry point.

I passi di costruzione di un control flow graph dal codice sono i seguenti:

- Individuazione dei basic blocks: per creare un basic block occorre capire dove inizia e dove finisce. Un blocco inizia in presenza di un statement leader che identifica la prima riga di un blocco. Uno statement è definito leader se è il primo statement del programma, se è lo statement obiettivo di un salto (condizionale o meno) oppure se segue immediatamente uno statement obbiettivo di un salto (condizionale o meno). Detto questo, un blocco inizia con uno statement leader e termina quando ne incontra un altro.
- Individuazione degli archi del grafo: gli archi devono essere posizionati in concomitanza di salti non condizionati, salti condizionati e salti sequenziali del programma per passare da blocco a blocco.

In figura vediamo un esempio di creazione di CFG (in rosso i numeri che identificano i basic blocks):



2.3 Analisi di Liveness

L'analisi di liveness è un'analsi di tipo *backward* (si esegue dal basso del CFG verso l'alto, quindi all'indietro) e *possible* (si esegue l'unione dei successori di un nodo). Questo significa che è un'analisi che osserva il futuro e la variabile in questione deve essere utilizzata almeno una volta dopo una sua definizione per essere viva. L'analisi di liveness è essenziale per l'allocazione dei registri: due variabili contemporaneamente vive non possono occupare lo stesso registro, pertanto conoscere quali sono le variabili vive ne permette l'allocazione ottimale e questo è utile in contesti di ottimizzazione di codice. In sintesi, una variabile x è live se si trova tra la sua definizione e l'uso. Viceversa, x non è live o (dead) se viene ridefinita prima di un successivo

uso oppure non viene mai usata. In particolare, x è live all'uscita di un blocco se verrà usata successivamente. Possiamo dare anche una definizione lungo i cammini. In questo caso, una variabile è live su un cammino π (con definizione solo all'inizio) se il cammino non contiene ulteriori definizioni ed in mezzo e c'è almeno un uso. Possiamo anche scomporre tale cammino in π_1 e π_2 dove k, punto di programma tra π_1 e π_2, contiene l'uso di x e π_1 non contiene definizioni ulteriori.

Di seguito mostriamo l'equazione di punto fisso e le regole semantiche per tale tipologia di analisi in riferimento al linguaggio definito precedentemente:

EQUAZIONE PUNTO FISSO

$$LiveOut(n) = \begin{cases} \emptyset & \text{se } n = exit \\ \bigcup_{m \in Succ(n)} LiveIn(m) & \text{altrimenti} \end{cases}$$

$$LiveIn(n) = Use(n) \cup (LiveOut(n) \backslash VarKill(n))$$

$$LiveOut(n) = \bigcup_{m \in Succ(n)} Use(m) \cup (LiveOut(m) \backslash VarKill(m))$$

SEMANTICA

Dominio astratto = $\mathcal{P}(Var)$
$L \subseteq Var$

$$[\![;]\!]^\sharp L = L$$
$$[\![NonZero(e)]\!]^\sharp L = [\![Zero(e)]\!]^\sharp L = L \cup Var(e)$$
$$[\![x \leftarrow e]\!]^\sharp L = Var(e) \cup (L \backslash \{x\})$$

$LiveIn(n)$ = {sono le variabili *live* in n che sono *live* su almeno un arco entrante}
$LiveOut(n)$ = {sono le variabili *live* in n che sono *live* su almeno un arco uscente}
$VarKill(n) = Def(n)$, cioè le definizioni presenti in n

3 Seconda parte: Progetto

Come già anticipato è stata utilizzata la libreria JgraphT che rende disponibili diverse tipologie di grafi con creazione parser in JavaCC per riconoscere istruzioni da codice e creare il CFG. Poi si sono create classi in Java che, dato grafo diretto, eseguono l'analisi di liveness. Da ultimo, utilizzo di Java Swing per l'interfaccia grafica.

3.1 JGraphT

Per la costruzione della componente principale utile all'analisi di liveness, ovvero il Control Flow Graph, si è deciso di guardare allo stato dell'arte in materia e la scelta è ricaduta sulla libreria JGraphT integrabile in Java e sviluppata da terzi. Tale libreria permette un'ampia gestione per quanto riguarda la costruzione di alberi o grafi. Tra le varie opzioni si è scelto di procedere con la costruzione di un DefaultDirectedGraph. Di default i nodi presenti non si prestavano allo scopo, pertanto si è proceduto con lo sviluppo e l'implementazione di un nuovo nodo. Il nodo creato è una classe Java che riporta metodi essenziali come i get e set e ha 2 attributi: istruzione e indice. Tali nodi verrano generati passo passo durante il processo di parsing ed aggiunti al grafo creato. A tal proposito il grafo dichiarato sarà formato da insieme di nodi definiti ad hoc e insieme di archi diretti definiti di default per la classe.
La dichiarazione del grafo è così fatta:

```
1 public static Graph<Vertexgraph,DefaultEdge> g = new
      DefaultDirectedGraph<Vertexgraph,DefaultEdge>(DefaultEdge.class);
```

3.2 Parser in JavaCC

Dati i pre-requisiti descritti in precedenza, per poter leggere il codice in input si è deciso di costruire un parser con JavaCC che supportasse comodamente il linguaggio del resto del progetto. Dalla schermata di inserimento del codice, l'interfaccia passa il testo scritto come input per il parser. Il parser controlla se il testo del codice soddisfa i pre-requisiti e che sia corretto sintatticamente rispetto alle usuali regole di un linguaggio di programmazione. Nel leggere una riga identifica le istruzioni costruendo i relativi nodi nel Control Flow Graph. Il CFG viene quindi costruito mentre il parser interpreta il codice.

3.2.1 Classe principale

La funzione *oneline()* crea una nuova istanza di Analisi il cui output sarà la String di output finale. La scelta di usare diverse variabili globali risponde a necessità implementative. Viene costruito inoltre il metodo **setIterations()** per settare successivamente nell'interfaccia il numero di iterazioni che si vorranno calcolare.

```
1  PARSER_BEGIN(Analisi)
2  package analisiProject;
3  import java.util.*;
4  import java.io.*;
5  import java.net.*;
6
7  import org.jgrapht.*;
8  import org.jgrapht.graph.*;
9  import org.jgrapht.io.*;
10 import org.jgrapht.traverse.*;
11
12 public class Analisi{
13
14    public void setIterations(int s) {
15       this.iterazioni = s;
16    }
17
18    public Analisi() {
19
20    }
21
22   public static Vertexgraph nodoPrec = new Vertexgraph("NULL",0);
23   public static Vertexgraph nodoWhile = null;
24   public static Vertexgraph nodoIf = null;
25   public static Vertexgraph nodoFor = null;
26   public static int i = 1;
27   public static int iterazioni=0;
28
29   public static Set<Vertexgraph> gen = new HashSet<Vertexgraph>();
30   public static Set<Vertexgraph> kill = new HashSet<Vertexgraph>();
31   public static Graph<Vertexgraph,DefaultEdge> g = new
         DefaultDirectedGraph<Vertexgraph,DefaultEdge>(DefaultEdge.class);
32
33    public String main(String args []) throws ParseException,
         FileNotFoundException {
34     Analisi P = new Analisi(new FileInputStream("corpoJava.txt"));
```

```
35
36      String riga = "" ;
37      String testo = "" ;
38      g.addVertex(nodoPrec);
39
40      try{
41        riga = P.one_line();
42       }
43       catch(Exception e){
44         System.out.println("NOK.");
45         System.out.println(e.getMessage());
46         e.printStackTrace();
47         }
48         return riga;
49    }
50  }
51
52  PARSER_END(Analisi)
```

3.2.2 Analizzatore Lessicale

Si riporta qui sotto l'elenco dei token utilizzati dal parser per la costruzione dei nodi del CFG. La sezione **SKIP** include i caratteri che vengono ignorati come ad esempio la dichiarazione di variabili intere.

```
1  SKIP :
2  {" "|"\r"|"\t"|"\n"|"int"}
3  TOKEN :{
4  < AND : "&&" > | < OR : "||" > | < NOT : "!" > | < DIMENSOPER : ">"
       | "<" | ">=" | "<=" >
5  |
6  < OPERATOR : "+" | "-" | "*" | "%" | "/" > | < ogr : "{" > | < cgr
       : "}" >
7  |
8  < FOR : "for" > | < WHILE :"while" | "While" | "WHILE" > | < IF
       :"if">| < NULL:"NULL" > | < EXIT:"EXIT" >|
9  < INTEGER : ([ "0"-"9" ])+ >
10 |
11 < END : ";" > | < OPENPAR : "(" > |< CLOSEPAR : ")" > | < EQUAL :
       "=" >| < COMMA : "," >|
12 < ANNCLASS : [ "A"-"Z","a"-"z" ]
13 | (["A"-"Z","a"-"z","_"]["A"-"Z","a"-"z","0"-"9","_" ])*) >}
```

3.2.3 Analizzatore Sintattico

Il metodo oneline() è il metodo principale invocato dal parser. Necessita di un oggetto Vertexgraph e di due stringhe di supporto.

```
String one_line() :
{
  Vertexgraph s;
  String u= "";
  String risultato="";
}

{
  (
    (
      s = simpleassign()
      {
        g.addVertex(s);
        g.addEdge(nodoPrec,s);
        nodoPrec = s;
        }
      |
      u = whilecycle()
      |
      u = ifcondition()
      |
      u = forcycle()
   )
  )*
   {
     risultato = "Insieme dei nodi: \n" +
         g.vertexSet().toString()+"\n"+
              "Insieme degli archi: \n" + g.edgeSet().toString()+"\n"+
              "Insieme delle var generate: \n"+gen.toString()+"\n"+
              "Insieme delle var killate: \n"+kill.toString()+"\n";

     Live obj = new Live(i, kill, gen, g);

     String results = obj.compute_LiveOut(iterazioni);

     return risultato+"\n"+results;
   }
}
```

Nel caso il parser incontri uno dei token per i costrutti (assegnamento semplice, ciclo while, if o ciclo for), chiamerà i relativi metodi. Al termine il risutato costrutito sarà una String ritornata come risultato. Quando il file in input viene esaminato tutto, *oneline()* crea un oggetto Live che computa l'analisi di liveness sul grafo appena generato e passato come parametro (tramite il metodo *computeLiveOut()*).

Gli insiemi GenSet e KillSet inizialmente vuoti vengono aggiornati ogni volta che una variabile viene incontrata. Sotto si riporta l'implementazione del metodo alla base della costruzione dei nodi, *simpleassign()*. Ogni volta che viene creato un nuovo nodo, un iteratore controlla se tale nodo è già presente o meno in uno degli insiemi.

```
1  Vertexgraph simpleassign() :
2  {
3    String s= "";
4    Token var=null;
5    Token value=null;
6    Token op=null;
7    String op1="";
8    Vertexgraph v = new Vertexgraph();
9  }
10
11 {
12   var = < ANNCLASS > < EQUAL >
13   {
14     s=(var.image);
15     Vertexgraph x = new Vertexgraph();
16     x.setNodo(s);
17     x.setIndex(i);
18     s=s+"=";
19
20    Iterator it = kill.iterator();
21    boolean presente=false;
22
23    while(it.hasNext()) {
24      Vertexgraph tmp = new Vertexgraph();
25      tmp = (Vertexgraph) it.next();
26      if(tmp.equals(x)) {
27        presente=true;
28        }
29    }
30
31    if(!presente) { kill.add(x);}
```

```
32    //if( ! kill.contains(x)) { kill.add(x);}
33   }
34    (
35      var = < ANNCLASS >
36      {
37        s = s+(var.image);
38       x = new Vertexgraph();
39        x.setNodo(var.image);
40        x.setIndex(i);
41        it = gen.iterator();
42       presente=false;
43
44        while(it.hasNext()) {
45          Vertexgraph tmp = new Vertexgraph();
46          tmp = (Vertexgraph) it.next();
47          if(tmp.equals(x)) {
48             presente=true;
49          }
50       }
51        if(!presente) { gen.add(x);}
52       //if(! gen.contains(x)) { gen.add(x);}
53      }
54      |
55      value = <INTEGER>
56      {
57        s=s+(value.image);
58      }
59    )
60    (
61     op= < OPERATOR >
62     {
63       s=s+(op.image);
64     }
65     (
66       var = < ANNCLASS >
67       {
68         Vertexgraph y = new Vertexgraph();
69         s = s+(var.image);
70         y.setNodo(var.image);
71         y.setIndex(i);
72         it = gen.iterator();
73         presente=false;
74
```

```
75          while(it.hasNext()) {
76            Vertexgraph tmp = new Vertexgraph();
77            tmp = (Vertexgraph) it.next();
78           if(tmp.equals(y)) {
79             presente=true;
80           }
81          }
82          if(!presente) { gen.add(y);}
83          //if(! gen.contains(y)) { gen.add(y);}
84        }
85        |
86        value = <INTEGER>
87        {
88          s=s+(value.image);
89        }
90     )
91    )* <END >
92     {
93       v.setNodo(s);
94       v.setIndex(i);
95       i = i+1;
96       return v;
97      }
98   }
```

Tale funzione supporta nodi costituiti come espressioni aritmetiche tra variabili e/o interi, aggiungendo via via le variabili agli insiemi di Gen e Kill. La classe simpleassign() istanzia nuovi Vertexgraph nel caso l'istruzione sia complessa creando quindi nuovi nodi e aggiungendoli direttamente al CFG. La variabile 'i' globale serve per tenere traccia dell'indice dei nodi, al termine dell'assegnamento vengono settati l'indice e l'istruzione.

3.3 Altre classi implementate

Per poter sfruttare al meglio la classe JGraphT si è resa necessaria l'implementazione di nuovi nodi per il CFG. La classe Vertexgraph implementa dei nodi personalizzati dotati di 2 valori (String istruzione e int indice). Il metodo più importante è la ridefinizione di *equals()* che servirà successivamente per controllare se un nodo è presente nell'insieme Gen e Kill.

3.3.1 Vertexgraph.java

```
public class Vertexgraph {
   public String nodo =""; //istruzione
   public int index= 0;

   public Vertexgraph() {
      nodo="NULL";
      index=-1;
   }

   public Vertexgraph(String istruzione, int indice) {
      nodo = istruzione;
      index = indice;
   }

   public String getIstruzione() {
      return nodo;
   }

   public int getIndex() {
      return index;
   }

   public void setNodo(String nodo) {
      this.nodo = nodo;
   }

   public void setIndex(int indice) {
      this.index = indice;
   }

   public String toString() {
      return "["+index+","+nodo+"]";
   }

   public boolean equals(Vertexgraph e) {
      return this.getIndex()==e.getIndex() &&
            this.getIstruzione().equals(e.getIstruzione();
   }
}
```

3.3.2 Live.java

La classe *Live.java* è la componente del progetto che computa l'analisi di liveness a partire dal CFG costruito dal parser precedentemente. Ciascuna iterazione viene supportata da due ArrayList<Set> in quanto la prima iterazione viene settata tutta vuota di default, mentre la successiva deve appoggiarsi alla precedente per il calcolo delle variabili. Al termine di ogni iterazione i due ArrayList<Set> vengono aggiornati. L'output parziale viene salvato su una stringa, al termine delle operazioni la matrice di LiveOut verrà stampata nell'interfaccia per "colonne" ovvero iterazioni.

```
1  package analisiProject;
2  import java.util.*;
3  import java.io.*;
4  import java.net.*;
5  import org.jgrapht.*;
6  import org.jgrapht.graph.*;
7  import org.jgrapht.io.*;
8  import org.jgrapht.traverse.*;
9
10 public class Live{
11
12    private Graph<Vertexgraph,DefaultEdge> g;
13    private Set<Vertexgraph> generated;
14    private Set<Vertexgraph> killed;
15    private int righe;
16    private String results ="";
17
18    private List<Set<String>> LiveOutCorr;
19
20
21    public Live(int num_nodi, Set<Vertexgraph> Kill, Set<Vertexgraph>
          Gen, Graph<Vertexgraph,DefaultEdge> g) {
22
23       righe = num_nodi;
24
25       killed = new HashSet<Vertexgraph>(Kill);
26       generated = new HashSet<Vertexgraph>(Gen);
27       this.g= new
             DefaultDirectedGraph<Vertexgraph,DefaultEdge>(DefaultEdge);
28       this.g = g;
29
30       LiveOutCorr = new ArrayList<Set<String>>(righe);
```

```
31
32      Set<String> tmp = new HashSet<String>();
33
34      for(int i = 0; i<righe; i++) {
35         this.LiveOutCorr.add(tmp);
36      }
37      results = "LIVE 0:\n";
38      for(int i= this.LiveOutCorr.size()-1; i>=0; i--) {
39         results= results+i+" "+LiveOutCorr.get(i)+"\n";
40
41      }
42
43   }//fine costruttore
44
45
46   public String compute_LiveOut(int iterazioni) {
47
48      for(int rep = 1; rep < iterazioni; rep++) {
49
50         List<Set<String>> LiveOutNext = new
               ArrayList<Set<String>>(righe); //colonna succ
51
52         Set<String> tmp = new HashSet<String>();
53
54         for(int i = 0; i<righe; i++) {
55            LiveOutNext.add(tmp);
56         }
57
58         Set<Vertexgraph> nodi = g.vertexSet();
59         Iterator itNodi = nodi.iterator();
60
61         while(itNodi.hasNext()) {//itero tutti i nodi
62
63            Vertexgraph curr = (Vertexgraph) itNodi.next();
64
65            List<Vertexgraph> uscenti = new ArrayList<Vertexgraph>();
66            uscenti=Graphs.successorListOf(g, curr);
67
68            Iterator itSucc = uscenti.iterator();
69
70            int indiceSucc = 0;
71
72            Set<String> liveOutSucc = new HashSet<String>();//live
```

```
            out del succ
73
74          while(itSucc.hasNext()) {
75
76             Set<String> genSucc = new HashSet<String>();//
                   generate dal succ
77             Set<String> killSucc = new HashSet<String>();
                   //killate dal succ
78
79             Vertexgraph succ = (Vertexgraph) itSucc.next();
80
81             indiceSucc = succ.getIndex();
82             Iterator gen = generated.iterator();
83
84             while(gen.hasNext()) {
85                Vertexgraph nodo = (Vertexgraph) gen.next();
86
87                if(nodo.getIndex()==indiceSucc) {
88                   genSucc.add(nodo.getIstruzione());//salvo i gen
                         dei successori
89                }
90             }
91
92             Iterator kill = killed.iterator();
93
94             while(kill.hasNext()) {
95                Vertexgraph nodo = (Vertexgraph) kill.next();
96                if(nodo.getIndex()==indiceSucc) {
97                   killSucc.add(nodo.getIstruzione());
98                }
99             }
100            //HO GEN E KILL PER i-esimo successore di CURR
101            liveOutSucc.addAll(this.LiveOutCorr.get(indiceSucc));
102
103            liveOutSucc.removeAll(killSucc);
104
105            liveOutSucc.addAll(genSucc);
106
107         }//fine while sui nodi uscenti
108
109         Set<String> union = new HashSet<String>(
               LiveOutNext.get(curr.getIndex()) );
110
```

```
111             union.addAll(liveOutSucc);
112
113             LiveOutNext.remove( curr.getIndex() );
114             LiveOutNext.add( curr.getIndex(), union );
115
116          }//fine iterazione sui nodi
117
118          results = results+"\n\nLIVE "+(rep)+":";
119          results=results+"\n";
120
121          for(int i= LiveOutNext.size()-1; i>=0; i--) {
122             results = results+i+" "+LiveOutNext.get(i)+"\n";
123          }
124
125          Collections.copy(this.LiveOutCorr,LiveOutNext);
126       }//fine for
127
128       return results;
129    }
130 }
```

4 Esecuzione

Per lanciare l'applicazione occorre essere in ambiente Eclipse e lanciare lo script principale *MainLauncher.java* che lancerà la classe principale di comunicazione con l'utente e provvederà poi all'analisi.

Il core principale della classe è il seguente:

```
1  JButton StartAnalysis = new JButton("Start analyze");
2       StartAnalysis.addActionListener(new ActionListener() {
3          public void actionPerformed(ActionEvent e) {
4             testo = textArea.getText();
5             iterazioni = Integer.parseInt(textArea_1.getText());
6             analisi.setIterations(iterazioni);
7
8             FileWriter fw;
9             try {
10                fw = new
                      FileWriter("utente/java/framework/corpoJava.txt");
11               fw.write(testo);
12               fw.flush();
13               fw.close();
```

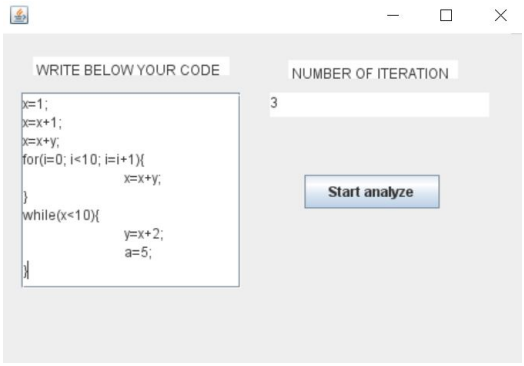
```
14            } catch (IOException e1) {
15               // TODO Auto-generated catch block
16               e1.printStackTrace();
17            }
18            try {
19               risultato = analisi.main(null);
20            } catch (FileNotFoundException e2) {
21               // TODO Auto-generated catch block
22               e2.printStackTrace();
23            } catch (ParseException e2) {
24               // TODO Auto-generated catch block
25               e2.printStackTrace();
26            }
27
28            JTextPane textpane = new JTextPane();
29            textpane.setBackground(Color.WHITE);
30            textpane.setText(risultato);
31            textpane.setFont(new Font("Arial", 0, 20));
32            textpane.setBounds(0,0, 1000,1000);
33            textpane.setEditable(false);
34
35            JScrollPane scroll = new JScrollPane(textpane);
36            scroll.setBounds(0, 0, 455, 300);
37
38              JFrame frame2 = new JFrame();
39              frame2.setBounds(100, 150, 450, 300);
40            frame2.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
41
42            frame2.getContentPane().add(scroll);
43             frame2.setLocationRelativeTo ( null );
44             frame2.setVisible ( true );
45            frame.dispose();
46         }
47      });
```

In particolare, alla riga 21 viene lanciato il file di parsing che a sua volta istanzierà una classe di analisi di Liveness e completerà l'analisi stampando il risultato in un *JScrollPane*.

4.1 Interfaccia Input/Output

Di seguito vengono riportate le interfacce di input e i risultati in output:

INTERFACCIA INPUT



CREAZIONE CFG E GENSET + KILLSET

Insieme dei nodi:
[[0,NULL], [1,x=1], [2,x=x+1], [3,x=x+y], [4,i=0], [5,i<10], [6,x=x+y], [7,i=i+1], [8,x<10], [9,y=x+2], [10,a=5]]
Insieme degli archi:
[([0,NULL] : [1,x=1]), ([1,x=1] : [2,x=x+1]), ([2,x=x+1] : [3,x=x+y]), ([3,x=x+y] : [4,i=0]), ([4,i=0] : [5,i<10]), ([5,i<10] : [6,x=x+y]), ([6,x=x+y] : [7,i=i+1]), ([7,i=i+1] : [5,i<10]), ([5,i<10] : [8,x<10]), ([8,x<10] : [9,y=x+2]), ([9,y=x+2] : [10,a=5]), ([10,a=5] : [8,x<10])]
Insieme delle var generate:
[[6,x], [8,x], [7,i], [5,i], [9,x], [2,x], [3,x], [6,y], [3,y]]
Insieme delle var killate:
[[3,x], [4,i], [2,x], [7,i], [1,x], [6,x], [9,y], [10,a]]

LIVEOUT

LIVE 0:	LIVE 1:	LIVE 2:
10 []	10 [x]	10 [x]
9 []	9 []	9 [x]
8 []	8 [x]	8 [x]
7 []	7 [i]	7 [x, y, i]
6 []	6 [i]	6 [i]
5 []	5 [x, y]	5 [x, i, y]
4 []	4 [i]	4 [x, y, i]
3 []	3 []	3 []
2 []	2 [x, y]	2 [x, y]
1 []	1 [x]	1 [x, y]
0 []	0 []	0 []