Taller #1 - SOOT

Ingeniería de Software II

Alumno: Lucas Raposeiras
LU: 34/15

Introducción

Los archivos generados por soot con los comandos descritos en el archivo `readme.txt` están en formato Jimple. Según Wikipedia, el formato Jimple es una representación intermedia de un programa Java diseñado para ser más fácil de optimizar que el bytecode de Java. Está escrito, tiene una sintaxis concreta y se basa en un código de tres direcciones. En especial, cabe destacar que **una línea de código en Java a veces es representada con más de una línea de código en Jimple**.

Ejercicio 1

Primero, presentamos el código del ejercicio:

```
    public class A {
        public static int main(String[] args) {
1:          int a = 8;
2.          if (args.length > 2)
3:              a = 5;
4:          int c = 1;
5:          while (!(c > a)) {
6:              c = c + c;
7:          }
8:          a = c - a;
9:          c = 0;
            return a + c;
        }
    }
```

Luego de compilar el código y ejecutar Soot según lo descrito en el archivo `readme.txt`, obtenemos un archivo Jimple.

Lo primero que vemos cuando analizamos el Jimple, es que:

1. Tanto las **líneas de código** como el **conjunto de hechos de la línea** generado por el *tagger* están escritos como un comentario, entre los caracteres `/*` y `*/`.
2. Las líneas de código marcadas en el Jimple no se corresponden exactamente con las líneas de código del código original en Java. Esto se debe a que Jimple es un código intermedio generado a partir del código Java original, como mencionamos en la introducción de este taller.

Leyendo el archivo Jimple particular de este ejercicio y considerando la segunda observación, detectamos las siguientes relaciones entre las líneas de código en Java y las líneas de código en Jimple:

- ❏ La línea **5** del código se corresponde con la línea **7** del código Jimple.
- ❏ La línea **8** del código se corresponde con la línea **10** del código Jimple.
- ❏ La línea **6** del código se corresponde con la línea **8** del código Jimple.

A continuacion extraeremos y analizaremos los fragmentos particulares de las líneas 7, 10 y 8 del archivo Jimple.

a) Línea 7 (Jimple) / Línea 5 (Java)

```
     label2:
        if c > a goto label3;
/*7*/
/*c has reaching def: c = 1*/
/*c has reaching def: c = c + c*/
/*a has reaching def: a = 8*/
/*a has reaching def: a = 5*/
```

Soot nos indica que en la línea 7 (es decir, línea 5 del código en Java) se alcanzan las siguientes definiciones:

➔ `c = 1`

Esta definición se corresponde con la línea 4 del código en Java. Soot reconoce que la línea 4 se ejecuta siempre, y como existe un camino entre la línea 4 y la línea 5 sin sobreescrituras de la variable `c`, afirma que `c = 1` es alcanzado en la línea 5.

➔ `c = c + c`

Esta definición se corresponde con la línea 6 del código en Java. Como el análisis que hace Soot sacrifica completeness (al ser un análisis estático de dataflow), se abstraen las dos ramas del if del while con una elección no determinista, con lo cual Soot supone que tanto la rama verdadera como la falsa se pueden evaluar. Dicho esto, Soot considera que la línea 6 se puede evaluar, y como existe un camino entre la línea 6 y la línea 5 sin sobreescrituras de la variable `c`, considera que `c = c + c` es alcanzado en la línea 5.

➔ `a = 8`

Esta definición se corresponde con la línea 1 del código en Java. Soot reconoce que la línea 1 se ejecuta siempre, y como existe un camino entre la línea 1 y la línea 5 sin sobreescrituras de la variable `a`, afirma que `c = 1` es alcanzado en la línea 5.

➔ `a = 5`

Esta definición se corresponde con la línea 3 del código en Java. Como el análisis que hace Soot sacrifica completeness (al ser un análisis estático de dataflow), se abstraen las dos ramas del if del while con una elección no determinista, con lo cual Soot supone que tanto la rama verdadera como la falsa se pueden evaluar. Dicho esto, Soot considera que la línea 3 se puede evaluar, y como existe un camino entre la línea 3 y la línea 5 sin sobreescrituras de la variable `a`, considera que `c = c + c` es alcanzado en la línea 5.

b) Línea 10 (Jimple) / Línea 8 (Java)

```
     label3:
        a#7 = c - a;
/*10*/
/*c has reaching def: c = 1*/
/*c has reaching def: c = c + c*/
/*a has reaching def: a = 8*/
/*a has reaching def: a = 5*/
```

Soot nos indica que en la línea 10 (es decir, línea 8 del código en Java) se alcanzan las siguientes definiciones:

➔ `c = 1`

Esta definición se corresponde con la línea 4 del código en Java. Soot reconoce que la línea 4 se ejecuta siempre, y como existe un camino entre la línea 4 y la línea 8 sin sobreescrituras de la variable `c`, afirma que `c = 1` es alcanzado en la línea 8.

➔ `c = c + c`

Esta definición se corresponde con la línea 6 del código en Java. Como el análisis que hace Soot sacrifica completeness (al ser un análisis estático de dataflow), se abstraen las dos ramas del if del while con una elección no determinista, con lo cual Soot supone que tanto la rama verdadera como la falsa se pueden evaluar. Dicho esto, Soot considera que la línea 6 se puede evaluar, y como existe un camino entre la línea 6 y la línea 8 sin sobreescrituras de la variable `c`, considera que `c = c + c` es alcanzado en la línea 8.

➔ `a = 8`

Esta definición se corresponde con la línea 1 del código en Java. Soot reconoce que la línea 1 se ejecuta siempre, y como existe un camino entre la línea 1 y la línea 8 sin sobreescrituras de la variable `a`, afirma que `c = 1` es alcanzado en la línea 8.

➔ `a = 5`

Esta definición se corresponde con la línea 3 del código en Java. Como el análisis que hace Soot sacrifica completeness (al ser un análisis estático de dataflow), se abstraen las dos ramas del if del while con una elección no determinista, con lo cual Soot supone que tanto la rama verdadera como la falsa se pueden evaluar. Dicho esto, Soot considera que la línea 3 se puede evaluar, y como existe un camino entre la línea 3 y la línea 8 sin sobreescrituras de la variable `a`, considera que `c = c + c` es alcanzado en la línea 8.

c) Línea 8 (Jimple) / Línea 6 (Java)

```
        c = c + c;
/*8*/
/*c has reaching def: c = 1*/
/*c has reaching def: c = c + c*/
/*c has reaching def: c = 1*/
/*c has reaching def: c = c + c*/
```

Soot nos indica que en la línea 8 (es decir, línea 6 del código en Java) se alcanzan las siguientes definiciones:

➔ `c = 1`

Esta definición se corresponde con la línea 4 del código en Java. Soot reconoce que la línea 4 se ejecuta siempre, y como existe un camino entre la línea 4 y la línea 6 sin sobreescrituras de la variable `c`, afirma que `c = 1` es alcanzado en la línea 6.

➔ `c = c + c`

Esta definición se corresponde con la línea 6 del código en Java. Como el análisis que hace Soot sacrifica completeness (al ser un análisis estático de dataflow), se abstraen las dos ramas del if del while con una elección no determinista, con lo cual Soot supone que tanto la rama verdadera como la falsa se pueden evaluar. Dicho esto, Soot considera que la línea 6 se puede evaluar, y como existe un camino entre la línea 6 y la línea 6 sin sobreescrituras de la variable `c`, considera que `c = c + c` es alcanzado en la línea 6.

Probablemente el análisis de Reaching Definitions arroja el resultado repetido por aparecer la variable `c` más de una vez en la línea de código 6.

Ejercicio 2

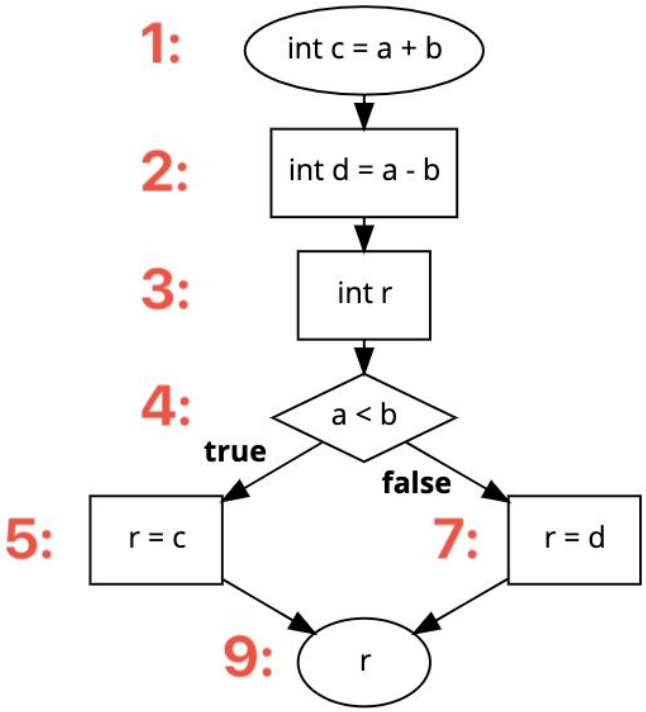
Primero, presentamos el código del ejercicio:

```
    public class B {
        public int exercise2(int a, int b) {
1:          int c = a + b;
2.          int d = a - b;
3:          int r;
4:          if (a < b) {
5:              r = c;
6:          } else {
7:              r = d;
8:          }
9:          return r;
        }
    }
```

Haciendo un análisis preliminar similar al del ejercicio anterior, notamos que:

- ❏ La línea **5** del código se corresponde con la línea **7** del código Jimple.
- ❏ La línea **7** del código se corresponde con la línea **9** del código Jimple.
- ❏ La línea **9** del código se corresponde con la línea **11** del código Jimple.

Además, observamos que salida del *Live Variables Tagger* para cada línea parece corresponderse con el conjunto de hechos *OUT* de dicha línea. Con lo cual, como en este ejercicio nos interesa el valor del conjunto *IN* de las líneas mencionadas, deberemos observar los conjunto OUT de las líneas predecesoras a estas, ya que en *Live Variables*, el conjunto *IN* de un nodo es la unión de los conjuntos *OUT* de sus nodos predecesores.



a) Para la línea 5 tenemos que la única línea predecesora es la línea 4. En el archivo Jimple vemos el siguiente resultado para la línea 4:

```
        if a >= b goto label1;
/*6*/
/*Live Variable: c*/
/*Live Variable: d*/
```

Con lo cual, podemos afirmar que el conjunto de variables vivas en la línea 5 es { `c`, `d` }.

b) Para la línea 7, tenemos que la única línea predecesora es la línea 4. En el archivo Jimple vemos el siguiente resultado para la línea 4:

```
        if a >= b goto label1;
/*6*/
/*Live Variable: c*/
/*Live Variable: d*/
```

Con lo cual, podemos afirmar que el conjunto de variables vivas en la línea 7 es { `c`, `d` }.

c) Para la línea 9, tenemos que las líneas predecesoras son las líneas 5 y 7. En el archivo Jimple vemos los siguientes resultados para las líneas 5 y 7:

Línea 7 (Jimple) / Línea 5 (Java)

```
        r = c;
/*7*/
/*Live Variable: r*/
```

Línea 9 (Jimple) / Línea 7 (Java)

```
     label1:
        r = d;
/*9*/
/*Live Variable: r*/
```

Con lo cual, podemos afirmar que el conjunto de variables vivas en la línea 9 es { `r` }.

Ejercicio 2

Primero, presentamos el código del ejercicio:

```
    public class C {
        private static class Cell {
            int value;
        }

        public int exercise3(Cell c1, Cell c2) {
1:          c1.value = 1;
2:          c2.value = 2;
3:          return c1.value;
        }
    }
```

Haciendo un análisis preliminar similar al de los ejercicios anteriores, notamos que:

> ❏ La línea `3` del código se corresponde con la línea `10` del código Jimple.

El análisis de *Null Pointer checker* nos dice si una línea de código tiene el potencial de levantar una `NullPointerException`, y agrega anotaciones que indican si el puntero que se desreferencia puede determinarse estáticamente o no como nulo, lo cual nos puede servir como valor abstracto.

En el archivo Jimple vemos el siguiente resultado para la línea 3:

```
        $stack3 = c1.<C$Cell: int value>;
/*10*/
/*[not null]*/
```

Es decir, el análisis nos dice que la instrucción `return c1.value` **no** va a arrojar una `NullPointerException`. Esto se debe a que `c1.value` está definido al momento de ejecutar la línea 3 de código (se hace una asignación a `c1.value` en la primer línea de código). Con lo cual, las variables c1 y c2 pueden alcanzar el valor abstracto `not null` línea 3.