

Ingeniería del Software II

Taller #7 – Implementando Análisis de Dataflow

LEER EL ENUNCIADO COMPLETO ANTES DE ARRANCAR.

Fecha de entrega: 6 de Junio de 2024

Fecha de re-entrega: 20 de Junio de 2024 (no hay extensiones)

Introducción

Un **zero-analysis** es un *may forward dataflow analysis* cuyo objetivo es detectar si una variable puede ser (o no) cero durante la ejecución de un programa. El análisis utiliza un conjunto de tuplas para determinar el valor abstracto de cada variable. Si el conjunto no tiene tuplas definidas para una variable, significa que se desconoce su valor abstracto. Las tuplas tienen la siguiente forma:

- $\langle x, \text{ZERO} \rangle$ representa que la variable x vale cero.
- $\langle x, \text{NOT_ZERO} \rangle$ representa que la variable x no vale cero.

Por lo tanto, para un conjunto $IN[n]$ de un nodo n del control-flow graph, podemos interpretar el valor abstracto de una variable x del siguiente modo:

- $IN[n](x) = \perp$ (bottom) si el $IN[n]$ no contiene tuplas con x .
- $IN[n](x) = Z$ (zero) si $IN[n]$ únicamente contiene una sola tupla de x y es la tupla $\langle x, \text{ZERO} \rangle$.
- $IN[n](x) = NZ$ (not_zero) si $IN[n]$ únicamente contiene sola tupla de x y es la tupla $\langle x, \text{NOT_ZERO} \rangle$.
- $IN[n](x) = MZ$ (maybe_zero) si $IN[n]$ contiene simultáneamente las tuplas $\langle x, \text{ZERO} \rangle$ y $\langle x, \text{NOT_ZERO} \rangle$.

Del mismo modo podemos interpretar los valores posibles de x para un conjunto $OUT[n]$.

Las ecuaciones de dataflow que caracterizan este análisis son las siguientes

$$IN[n] = \bigcup_{n' \in \text{pred}(n)} OUT[n']$$

$$OUT[n] = \text{Transfer}[n, IN[n]]$$

Ejercicios

Parte 1: Definiendo el Zero Analysis

Cada ejercicio se relaciona con una función del código que se verá en la parte dos, por ahora puede desestimar los comentarios de *Función*.

Sean $x, y, z \in \mathbb{Z}$

Ejercicio 1

Función `ZeroValueVisitor::visitIntegerConstant`

Completar la siguiente tabla con los valores esperados del conjunto $OUT[n]$ para la variable x cuando n es la asignación de una constante, con $K \in \mathbb{Z} - \{0\}$:

n	OUT[n](x)
x = 0	
x = K // con K distinto de 0	

Ejercicio 2

Función ZeroValueVisitor::visitLocal

Completar la siguiente tabla para caracterizar la función de transferencia (*Transfer*) para la sentencia de copia ($x = y$):

IN[n](y)	OUT[n](x)
\perp	
Z	
NZ	
MZ	

Ejercicio 3

Función ZeroAbstractValue::add

Completar la siguiente tabla para caracterizar la función de transferencia (*Transfer*) cuando se calcula la información de dataflow para una sentencia de suma ($x = y + z$):

IN[n](y)	IN[n](z)	OUT[n](x)
\perp	\perp	
Z	\perp	
NZ	\perp	
MZ	\perp	
\perp	Z	
Z	Z	
NZ	Z	
MZ	Z	
\perp	NZ	
Z	NZ	
NZ	NZ	
MZ	NZ	
\perp	MZ	
Z	MZ	
NZ	MZ	
MZ	MZ	

Ejercicio 4

Función ZeroAbstractValue::subtract

Completar la siguiente tabla para caracterizar la función de transferencia (*Transfer*) cuando se calcula la información de dataflow para una sentencia de resta ($x = y - z$):

IN[n](y)	IN[n](z)	OUT[n](x)
\perp	\perp	
Z	\perp	
NZ	\perp	
MZ	\perp	
\perp	Z	
Z	Z	
NZ	Z	
MZ	Z	
\perp	NZ	
Z	NZ	
NZ	NZ	
MZ	NZ	
\perp	MZ	
Z	MZ	
NZ	MZ	
MZ	MZ	

Ejercicio 5

Función ZeroAbstractValue::multiplyBy

Completar la siguiente tabla para caracterizar la función de transferencia (*Transfer*) cuando se calcula la información de dataflow para una sentencia de multiplicación ($x = y * z$):

IN[n](y)	IN[n](z)	OUT[n](x)
\perp	\perp	
Z	\perp	
NZ	\perp	
MZ	\perp	
\perp	Z	
Z	Z	
NZ	Z	
MZ	Z	
\perp	NZ	
Z	NZ	
NZ	NZ	
MZ	NZ	
\perp	MZ	
Z	MZ	
NZ	MZ	
MZ	MZ	

Ejercicio 6

Función ZeroAbstractValue::divideBy

Completar la siguiente tabla para caracterizar la función de transferencia (*Transfer*) cuando se calcula la información de dataflow para una sentencia de división **entera** ($x = y / z$):

IN[n](y)	IN[n](z)	OUT[n](x)
\perp	\perp	
Z	\perp	
NZ	\perp	
MZ	\perp	
\perp	Z	
Z	Z	
NZ	Z	
MZ	Z	
\perp	NZ	
Z	NZ	
NZ	NZ	
MZ	NZ	
\perp	MZ	
Z	MZ	
NZ	MZ	
MZ	MZ	

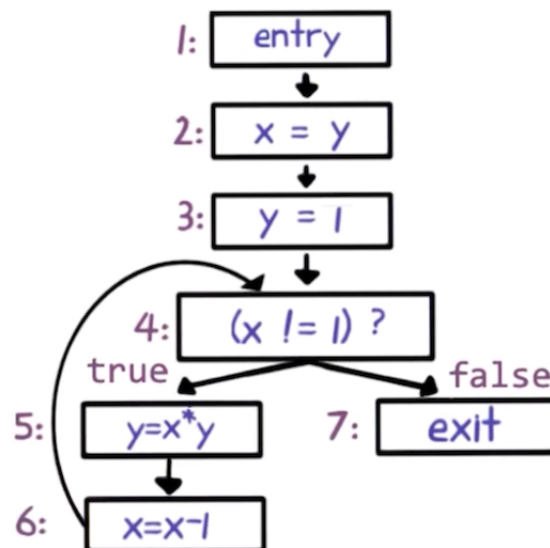
Ejercicio 7

Sea el siguiente programa y su correspondiente control-flow graph:

```

public int productoria(int y) {
    int x = y;
    y = 1;
    while (x != 1) {
        y = x * y;
        x = x - 1;
    }
    return y;
}

```



Ejecutar el algoritmo caótico iterativo hasta obtener valores estables de *IN* y *OUT* para toda variable del programa. **Nótese que el valor inicial de los parámetros es siempre MZ ya que no sabemos que valor pueden tomar al ejecutarse el programa.**

n	IN[n](x)	IN[n](y)	OUT[n](x)	OUT[n](y)
1	\perp	MZ		
2				
3				
4				
5				
6				
7				

Parte 2: Implementando el Zero Analysis en SOOT

Para este taller se pide implementar un dataflow analysis que infiere información sobre el valor de una variable entera con el fin de detectar si hay una división por cero. Para este objetivo tendrán que completar una implementación ya existente que deberán bajar del campus de la materia. Al importar el proyecto en *IntelliJ IDEA*, puede hacer falta configurar el JDK de la siguiente manera: Settings > Build, Execution, Deployment > Build Tools > Gradle → Gradle JVM 1.8

La implementación debe soportar (al menos) las siguientes operaciones:

- $x = 0;$
- $x = K;$ // donde K es una constante distinta de cero
- $x = y;$
- $x = y + z;$
- $x = y - z;$
- $x = y * z;$
- $x = y / z;$ // división entera

Deberán completar los métodos del `enum ZeroAbstractValue`; los métodos `visitDivExpression` y `visitIntegerConstant` de la clase `ZeroValueVisitor`; el método `union` de la clase `ZeroAbstractSet` y (se recomienda utilizar el método anterior) el método `merge` de la clase `DivisionByZeroAnalysis`.

Algunos de los archivos y carpetas en el proyecto son:

- **targets** contiene las clases a ser analizadas en el Taller.
- **util** contiene la implementación del patrón de diseño visitor para visitar los distintos **statement** del programa.
- **zeroanalysis** es la carpeta donde está el código a modificar.
 - **Launcher:** Es el *entry point* del analizador.
 - **DivisionByZeroAnalysis:** es la clase que implementa el zero-analysis (extends `ForwardFlowAnalysis` de Soot).
 - **ZeroValueVisitor:** contiene la implementación del patrón de diseño visitor para visitar las distintas **expresiones** del programa (extends `AbstractValueVisitor` de utils).
 - **ZeroAbstractSet:** es un conjunto de tuplas cuya primer componente es el nombres de variable y la segunda componente es un `ZeroAbstractValue`.
 - **ZeroAbstractValue:** implementa los valores abstractos mencionados anteriormente.

- **TallerTest:** los tests que deberá pasar la implementación.

Para correr los tests del proyecto puede hacerlo desde una IDE como *IntelliJ IDEA* o desde la terminal con el comando `./gradlew test`. Si se desea correr el Division By Zero Analysis desde la terminal para una clase, se puede hacer con el comando `./gradlew zeroAnalysis <targetClass>`.

Recordar que para que este taller funcione se debe tener instalado Java SE Runtime Environment 8 (ver Taller anterior ante dudas).

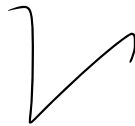
El análisis implementado debe calcular correctamente la información de dataflow como se muestra a continuación, indicando los posibles errores de división por cero si los hubiere:

```
public static int test1(int m, int n) {
    int x = 0;
    int k = x * n;
    int j = m / k;
    return j; // IN(x)=IN(k)=Z, IN(m)=IN(n)=MZ, IN(j)=Bottom
}
```

```
public static int test2(int m, int n) {
    int x = n - n;
    int i = x + m;
    int j = m / x;
    return j; // IN(m)=IN(n)=IN(x)=IN(i)=MZ, IN(j)=MZ
}
```

```
public static int test3(int m, int n) {
    int x = 0;
    int j = m / n;
    return j; // IN(m)=IN(n)=MZ, IN(x)=Z, IN(j)=MZ
}
```

```
public static int test4(int m, int n) {
    int x = 0;
    if (m != 0) {
        x = m;
    } else {
        x = 1;
    }
    int j = n / x;
    return j; // IN(m)=IN(n)=IN(x)=MZ, IN(j)=MZ
}
```



IN(X) = MZ
IN(Y) = NZ

```
public static int test5(int y) {
    int x = y;
    y = 1;
    while (x != 1) {
        y = x * y;
        x = x - 1;
    }
    return y; // IN(x)=IN(y)=MZ
}
```

IN(X) = MZ
IN(Y) = MZ
IN(X) = IN(Y) = MZ

```
public static int test6(int x) {  
    int y;  
    if (x == 0) {  
        y = 1;  
    } else {  
        y = 2;  
    }  
    int r = x / y;  
    return r; // IN(x)=IN(r)=MZ, IN(y)=NZ  
}
```

```
public static int test7() {  
    int i = 0, j = 1;  
    int d = j / i;  
    if (d > 0) {  
        d = 1;  
    }  
    return d; // IN(i)=Z, IN(j)=NZ, IN(d)=NZ  
}
```

X = MZ
Y = B
Y = NZ
r = M

Formato de Entrega

El taller debe ser subido al campus. Debe ser un archivo zip con el siguiente contenido.

1. Un archivo **respuestas** con las respuestas a los ejercicios de la Parte 1 (puede ser .md, .txt, .pdf, .png, .csv o .xlsx)
2. Una carpeta con la implementación completa y funcionando del Zero Analysis. Esta implementación debe pasar los tests del archivo **TallerTest.java**. Para los métodos modificados, agregue comentarios explicando la solución desarrollada. Por favor incluir el proyecto completo.