

Ingeniería del Software II

Taller #5 – Zero Analysis

LEER EL ENUNCIADO COMPLETO ANTES DE ARRANCAR.

Fecha de entrega: 28 de Mayo de 2025 (**16:59hs**)

Fecha de re-entrega: 11 de Junio 2025 (**16:59hs**) (**no hay extensiones**)

Introducción

Un **zero-analysis** es un *may forward dataflow analysis* cuyo objetivo es detectar si una variable de tipo entero puede ser cero (o no) durante la ejecución de un programa. El análisis representa al estado como una función de variables del programa a un reticulado de valores abstractos $Sign = \{\perp, +, 0, -, \top\}$.

Por lo tanto, el valor $IN[n]$ de un nodo n del control-flow graph, será un estado abstracto que lo podemos modelar del siguiente modo:

- $IN[n](x) = \perp$ (bottom) si x nunca tuvo un valor al entrar al nodo.
- $IN[n](x) = 0$ (zero) si el valor de x es siempre cero al entrar al nodo.
- $IN[n](x) = +$ (not_zero) si el valor de x es siempre positivo al nodo..
- $IN[n](x) = -$ (not_zero) si el valor de x es siempre negativo al nodo..
- $IN[n](x) = \top$ (maybe_zero) si x puede ser cualquier valor al nodo.

Del mismo modo podemos interpretar los valores posibles de x para el estado abstracto $OUT[n]$. Las ecuaciones de dataflow que caracterizan este análisis son las siguientes

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

$$OUT[n] = Transfer[n, IN[n]]$$

Parte 1: Definiendo el Zero Analysis

Los ejercicios de la parte 1 son para entrar en calor antes de comenzar con la implementación. No se entregan, pero se usan para poder consultar potenciales dudas acerca de la semantica del análisis.

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

Ejercicio 1

Función ZeroValueVisitor::visitIntegerConstant

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

n	OUT[n](x)
$x = 0$	
$x = K \ // \text{ con } K > 0$	
$x = K \ // \text{ con } K < 0$	
$x = K \ // \text{ con } K = 0$	

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	
...	

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	
...		

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	
...		

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	
...		

Ejercicio 6

Función ZeroAbstractValue::divideBy

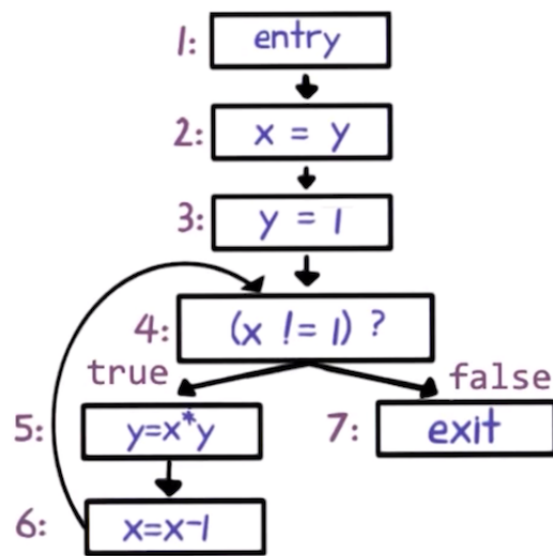
Completar la siguiente tabla para caracterizar la función de transferencia (*Transfer*) cuando se calcula la información de dataflow para una sentencia de división **entera** ($x = y / z$):

IN[n](y)	IN[n](z)	OUT[n](x)
\perp	\perp	
IN[n](y)	IN[n](z)	OUT[n](x)
\perp	\perp	
...		

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	\top		
2				
3				
4				
5				
6				
7				

Parte 2: Implementando el Zero Analysis en SOOT

Para que el taller funcione se debe tener instalado Java SE Runtime Environment 8.

Soot

La herramienta Soot contiene un conjunto de algoritmos ya implementados para *Dataflow Analysis*. Puede encontrar más información acerca de su funcionamiento, opciones y documentación en los siguientes links:

- Wiki oficial: <https://github.com/soot-oss/soot/wiki>
- Documentación de la API Java:
<https://soot-oss.github.io/soot/docs/4.4.1/jdoc/index.html>
- Documentación con las opciones para usar la herramienta por línea de comando:
https://soot-oss.github.io/soot/docs/4.4.1/options/soot_options.html

Implementación

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 `ZeroAbstractState` 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).
 - **ZeroAbstractState:** un mapping entre nombres de variable y 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>`.

El análisis implementado debe calcular correctamente la información de dataflow. Los test incluidos en el proyecto leen los programas en java incluidos en la carpeta `targets` y validan que los valores abstractos de las variables en determinados puntos del programa sean calculados correctamente y que se detecte una potencial división por cero.

Formato de Entrega

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

1. 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.
2. BONUS TRACK (opcional/**no entregable**/sin corrección): Modificar el reticulado para usar el reticulado de intervalos., reimplementando adecuadamente las operaciones. Modificar y/o agregar nuevos test si hace falta.