Ingeniería del Software II

Taller #1 – Zero Analysis

LEER EL ENUNCIADO COMPLETO ANTES DE ARRANCAR.

Fecha de entrega: 26 de Agosto de 2024 (23:55hs)
Fecha de re-entrega: 14 de Noviembre de 2024 (23:55hs) **(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 $L = \{\bot, ZERO, NZ, MZ\}$.

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:

- $IN[n](x) = \bot$ (bottom) si x nunca tuvo un valor al entrar al nodo.
- $IN[n](x) = Z$ (zero) si el valor de x es siempre cero al entrar al nodo.
- $IN[n](x) = NZ$ (not_zero) si el valor de x es siempre disinto a cero al entrar al nodo.
- $IN[n](x) = MZ$ (maybe_zero) si x puede ser un el valor cero o distinto a cero al entrar 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 **deben ser completados en el campus**, las tablas a continuación son solo de referencia, 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 estado abstracto $OUT[n]$ 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)
$\bot$	
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)
$\bot$	$\bot$	
Z	$\bot$	
NZ	$\bot$	
MZ	$\bot$	
$\bot$	Z	
Z	Z	
NZ	Z	
MZ	Z	
$\bot$	NZ	
Z	NZ	
NZ	NZ	
MZ	NZ	
$\bot$	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)
$\bot$	$\bot$	
Z	$\bot$	
NZ	$\bot$	
MZ	$\bot$	
$\bot$	Z	
Z	Z	
NZ	Z	
MZ	Z	
$\bot$	NZ	
Z	NZ	
NZ	NZ	
MZ	NZ	
$\bot$	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)
$\bot$	$\bot$	
Z	$\bot$	
NZ	$\bot$	
MZ	$\bot$	
$\bot$	Z	
Z	Z	
NZ	Z	
MZ	Z	
$\bot$	NZ	
Z	NZ	
NZ	NZ	
MZ	NZ	
$\bot$	MZ	
Z	MZ	
NZ	MZ	
MZ	MZ	

Ejercicio 6

Función ZeroAbstractValue::divideBy

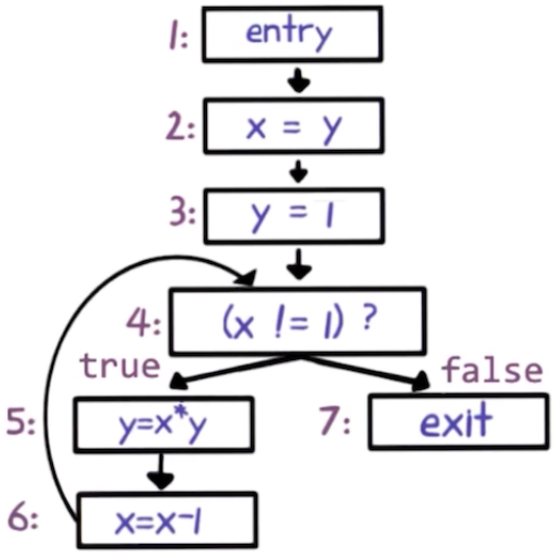
Completar la siguiente tabla para caracterizar la función de transferencia ($Transfer$) cuando se calcula la información de dataflow para una sentencia de división **entera** (`x = y / z`):

IN[n](y)	IN[n](z)	OUT[n](x)
$\bot$	$\bot$	
Z	$\bot$	
NZ	$\bot$	
MZ	$\bot$	
$\bot$	Z	
Z	Z	
NZ	Z	
MZ	Z	
$\bot$	NZ	
Z	NZ	
NZ	NZ	
MZ	NZ	
$\bot$	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 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 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
}

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
}

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
}
```

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 desarrolada. Por favor incluir el proyecto completo.

3. BONUS TRACK (opcional/no entregable/sin correción): Modificar el reticulado para usar el de *Sign* (o el *Sign* extendido con ≤ 0 y ≥ 0), reimplementando adecuadamente las operaciones. Modificar y/o agregar nuevos test si hace falta.