Taller #2 - Zero-Analysis

Ingeniería de Software II

Alumno: Lucas Raposeiras
LU: 34/15

Resolución

Clase `ZeroLattice`

Métodos auxiliares

Para aportar a legibilidad y expresividad del código, decidí crear los siguientes métodos de instancia auxiliares:

- ❏ **`private boolean isBottom()`**: Que devuelve **`true`** si el valor abstracto es `BOTTOM` y **`false`** en caso contrario.
- ❏ **`private boolean isNotZero()`**: Que devuelve **`true`** si el valor abstracto es `NOT_ZERO` y **`false`** en caso contrario.
- ❏ **`private boolean isZero()`**: Que devuelve **`true`** si el valor abstracto es `ZERO` y **`false`** en caso contrario.

Los métodos **`add`**, **`divideBy`**, **`multiplyBy`** y **`substract`** son ejecutados en la clase **`ZeroLatticeValueVisitor`** cuando el analizador visita una expresión de tipo suma, división, multiplicación o resta respectivamente, y se encargan de computar el valor abstracto resultante de la operación correspondiente.

El método **`supreme`** se ejecuta en la operación **`merge`** de la clase **`DivisionByZeroAnalysis`**, y se encarga de devolver el “mayor” valor abstracto entre dos según el modelo de reticulado.

- **Método `add`**

  Código:

```
public ZeroLattice add(ZeroLattice another) {
  if (this.isZero() && another.isZero()) {
     return ZeroLattice.ZERO;
  }

  if (this.isZero() && another.isNotZero() || this.isNotZero() && another.isZero()) {
     return ZeroLattice.NOT_ZERO;
  }

  return ZeroLattice.MAYBE_ZERO;
}
```

  Explicación:

  Devolveremos el valor abstracto:

  a. `ZERO` si **ambos** sumandos son `ZERO`, ya que sabemos que $0 + 0 = 0$.
  b. `NOT_ZERO` si un sumando es `ZERO` y el otro es `NOT_ZERO`, ya que al ser 0 el elemento neutro de la suma, al sumarse con un valor entero distinto de 0 obtenemos el mismo valor entero, el cual se corresponde con el valor abstracto `NOT_ZERO`.
  c. `MAYBE_ZERO` en otro caso, ya que no tenemos la certeza de qué valor abstracto tiene la variable.

- **Método `divideBy`**

  Código:

```
public ZeroLattice divideBy(ZeroLattice another) {
      if (this.isZero() && another.isNotZero()) {
            return ZeroLattice.ZERO;
      }

      if (this.isNotZero() && another.isNotZero()) {
            return ZeroLattice.NOT_ZERO;
      }

      return ZeroLattice.MAYBE_ZERO;
}
```

Explicación:

Devolveremos el valor abstracto:

a. `ZERO` si el dividendo es `ZERO` y el divisor es `NOT_ZERO`, ya que al dividir 0 por cualquier valor entero distinto de 0 obtenemos 0, el cual se corresponde con el valor abstracto `ZERO`.
b. `NOT_ZERO` si **ambos** valores son `NOT_ZERO`, ya que no podemos obtener 0 dividiendo dos valores distintos de 0.
c. `MAYBE_ZERO` en otro caso, ya que no tenemos la certeza de qué valor abstracto tiene la variable. Notar que esto incluye el caso de la división por 0, la cual atenderemos en la clase **`ZeroLatticeValueVisitor`**. Simplemente retornamos `MAYBE_ZERO` como convención.

- **Método `multiplyBy`**

  Código:

```
public ZeroLattice multiplyBy(ZeroLattice another) {
  if (this.isZero() || another.isZero()) {
     return ZeroLattice.ZERO;
  }

  if (this.isNotZero() && another.isNotZero()) {
     return ZeroLattice.NOT_ZERO;
  }

  return ZeroLattice.MAYBE_ZERO;
}
```

  Explicación:

  Devolveremos el valor abstracto:

  a. `ZERO` si **alguno** de los factores es `ZERO`, ya que al ser 0 el elemento absorbente de la multiplicación, al multiplicarse con cualquier valor entero obtenemos 0, el cual se corresponde con el valor abstracto `ZERO`.
  b. `NOT_ZERO` si **ambos** factores son `NOT_ZERO`, ya que no podemos obtener 0 multiplicando dos valores distintos de 0.
  c. `MAYBE_ZERO` en otro caso, ya que no tenemos la certeza de qué valor abstracto tiene la variable.

- **Método `substract`**

  Código:

```
public ZeroLattice substract(ZeroLattice another) {
  if (this.isZero() && another.isZero()) {
     return ZeroLattice.ZERO;
  }

  if (this.isZero() && another.isNotZero() || this.isNotZero() && another.isZero()) {
     return ZeroLattice.NOT_ZERO;
  }

  return ZeroLattice.MAYBE_ZERO;
}
```

  Explicación:

  Devolveremos el valor abstracto:

  a. `ZERO` si tanto el minuendo como el sustraendo son `ZERO`, ya que sabemos que $0 - 0 = 0$.
  b. `NOT_ZERO` si uno de los valores es `ZERO` y el otro es `NOT_ZERO`, ya que al ser 0 el elemento neutro de la resta, al restarle 0 a un valor entero distinto de 0 obtenemos el mismo valor entero, el cual se corresponde con el valor abstracto `NOT_ZERO`.
  c. `MAYBE_ZERO` en otro caso, ya que no tenemos la certeza de qué valor abstracto tiene la variable.

- **Método `supreme`**

  Código:

```
public ZeroLattice supreme(ZeroLattice another) {
  if (this.isBottom()) {
     return another;
  }

  if (another.isBottom()) {
     return this;
  }
  if (this.equals(another)) {
     return this;
  }

  return ZeroLattice.MAYBE_ZERO;
}
```

  Explicación:

  Dados los valores abstractos `this` y `another`, devolveremos:

  a. `another` si `this` es `BOTTOM`, ya que seguro `another` es “mayor” que `BOTTOM`, o en el peor caso, igual.
  b. `this` si `another` es `BOTTOM`, ya que seguro `this` es “mayor” que `BOTTOM`, o en el peor caso, igual.
  c. Cualquiera de los dos (`this` o `another`) si ambos son el mismo valor abstracto.
  d. `MAYBE_ZERO` en otro caso, ya que seguro que es el único valor abstracto “mayor” que `this` y `another`.

Clase `ZeroLatticeValueVisitor`

La clase **`ZeroLatticeValueVisitor`** contiene dos *campos* que nos interesan: `resolvedValue` y `possibleDivisionByZero`.

El campo `resolvedValue` es de tipo **`ZeroLattice`**, e indica el valor abstracto del nodo.

El campo `possibleDivisionByZero` es de tipo **`Boolean`**, y es utilizado para indicar que el nodo posiblemente realiza una división por cero. Si este campo es **`true`** para un nodo dado, Soot imprimirá el tag de texto *“Possible division by zero here”* debajo de dicho nodo.

- **Método `visitDivExpression`**

  El método **`visitDivExpression`** se ejecuta cuando una expresión es binaria y de tipo **división**.
  Toma dos argumentos: el valor abstracto computado para el **dividendo** y el valor abstracto computado para el **divisor**.

  Código:

```
public void visitDivExpression(ZeroLattice leftOperand, ZeroLattice rightOperand) {
	resolvedValue = leftOperand.divideBy(rightOperand);
	possibleDivisionByZero = rightOperand != ZeroLattice.NOT_ZERO;
}
```

  Explicación:

  De todos los métodos relacionados a expresiones aritméticas que implementa la clase **`ZeroLatticeValueVisitor`**, **`visitDivExpression`** es el único que nos interesa para poder hacer un análisis de división por cero. En `resolvedValue` asignamos el resultado abstracto de la división, mientras que en `possibleDivisionByZero` asignaremos **`false`** si el divisor es el valor abstracto `NOT_ZERO`, y **`true`** en caso contrario. Esto es así ya que el único caso donde sabemos con certeza que no se está dividiendo por cero es cuando en el dividendo tenemos el valor abstracto `NOT_ZERO`.

- **Método `visitIntegerConstant`**

  El método **`visitIntegerConstant`** se ejecuta cuando una expresión es una constante de tipo **`int`**.
  Toma como único argumento el valor de tipo `int` de la expresión.

  Código:

```
public void visitIntegerConstant(int value) {
	resolvedValue = value == 0
		? ZeroLattice.ZERO
		: ZeroLattice.NOT_ZERO;
}
```

  Explicación:

  Como la expresión que estamos evaluando en este método es una constante de tipo `int`, **tenemos mucha información**. En particular, sabemos el valor de dicha constante y por lo tanto podemos determinar el valor abstracto a resolver. Simplemente asignamos el valor abstracto `ZERO` a `resolvedValue` si el valor de la constante es cero y, en caso contrario, le asignamos el valor abstracto `NOT_ZERO`.

Clase `DivisionByZeroAnalysis`

- **Método `merge`**

  El método **`merge`** es ejecutado por soot para fusionar conjuntos de hechos de flujo en puntos de fusión del flujo de control, por ejemplo, al final de una declaración de ramificación (if/then/else).

  Toma tres argumentos: el conjunto de hechos de la **rama izquierda**, el conjunto de hechos de la **rama derecha** y otro conjunto, que será el conjunto de **hechos de salida**, el cual debemos modificar.

  Código:

```
protected void merge(VariableToLatticeMap input1, VariableToLatticeMap input2, VariableToLatticeMap output) {
	output.putAll(input1);

	input2.forEach((key, value) -> {
		output.put(key, output.containsKey(key) ? output.get(key).supreme(value) : value);
	});
}
```

  Explicación:

  Inicialmente se copian todos los pares (clave, valor) de `input1` en `output`. Luego, para cada par `(key, value)` de `input2`, se inserta dicho par en `output` si es que la clave `key` no está definida ya en `output`. En el caso donde la clave `key` ya está definida en `output` (por haber sido copiada desde `input1`), entonces se inserta en `output` el supremo entre los valores correspondientes a dicha clave en `output` y `input2`. Este caso representa que la misma variable tiene un valor abstracto asignado en ambas ramificaciones, con lo cual, resolvemos la disputa obteniendo el supremo de ellos. Cabe mencionar que `key` es un *string* que representa el nombre de la variable a la cual nos referimos, mientras que `value` es un objeto de tipo `VariableToLatticeMap` que representa un valor abstracto.

Resultados obtenidos

Ejercicio 1

```
    public int ejercicio1 (int m, int n) {
1:      int x = 0;
2:      int j = m / (x * n);
        /*Possible division by zero here*/
3:      return j;
    }
```

En este ejercicio el analizador etiquetó una posible división por cero en la línea 2.

Análisis realizado por el analizador:

1. En la línea 1, el valor abstracto de `x` es `ZERO`, ya que es una asignación de constante igual a 0.
2. En la línea 2, el valor abstracto de la expresión `x * n` es `ZERO`, ya que es una multiplicación donde uno de los factores (x) tiene valor abstracto `ZERO`.
3. **Por lo tanto, en la línea 2, el analizador marca una posible división por cero.**

Ejercicio 2

```
     public int ejercicio2 (int m, int n) {
1:         int x = n - n;
2:         int i = x + m;
3:         int j = m / x;
           /*Possible division by zero here*/
4:         return j;
     }
```

En este ejercicio el analizador etiquetó una posible división por cero en la línea 3.

Análisis realizado por el analizador:

1. En la línea 1, el valor abstracto de `x` es `MAYBE_ZERO`, ya que es una resta en la cual no se tiene información sobre `n`.
2. En la línea 2, no se modifica el valor abstracto de `x`.
3. En la línea 3, el valor abstracto de `x` es `MAYBE_ZERO`.
4. **Por lo tanto, en la línea 3, el analizador marca una posible división por cero.**

Ejercicio 3

```
     public int ejercicio3 (int m, int n) {
1:         int x = 0;
2:         if (m != 0) {
3:               x = m;
4:         } else {
5:               x = 1;
6:         }
7:         int j = n / x;
           return j;
8:   }
```

En este ejercicio el analizador no etiquetó ninguna posible división por cero. La única línea donde el analizador podría haber etiquetado una posible división por cero es en la línea 7. Veamos por qué el analizador no etiquetó una posible división por cero.

Análisis realizado por el analizador:

1. En la línea 1, el valor abstracto de `x` es `ZERO`, ya que es una asignación de constante igual a 0.
2. En la línea 3, en la rama verdadera del if, el valor abstracto de `x` es `BOTTOM`, ya que obtiene el mismo valor abstracto de `x`, del cual no se tiene información.
3. En la línea 5, en la rama falsa del if, el valor abstracto de `x` es `NOT_ZERO`, ya que es una asignación de constante distinta de 0.
4. En la línea 7, se realiza un merge entre el conjunto de hechos de la rama verdadera del if y el conjunto de hechos de la rama falsa del if.
5. En la línea 7, el valor abstracto de `x` es `NOT_ZERO`, ya que es el supremo entre `BOTTOM` y `NOT_ZERO`.
6. **Por lo tanto, en la línea 7, el analizador no marca una posible división por cero.**

Ejercicio 4

```
     public int ejercicio4 (int m, int n) {
1:         int x = 0;
2:         int j = m / n;
           /*Possible division by zero here*/
3:         return j;
     }
```

En este ejercicio el analizador etiquetó una posible división por cero en la línea 2.

Análisis realizado por el analizador:

1. En la línea 2, el valor abstracto de `n` es `BOTTOM`, ya no se tiene información sobre `n`.
2. **Por lo tanto, en la línea 2, el analizador marca una posible división por cero.**