AAP - Trabajo Práctico 2: Análisis Dataflow
Detección de accesos ilegales a *arrays*

Bugni , Paulovsky , Perez (Grupo 2)

November 2, 2011

1 Ejercicio 1

Se realizaron los siguientes cambios en el código para detectar posibles errores de referencias nulas:

- Agregamos soporte para parámetros que sean **objetos** arbitrarios. Hecho ese cambio la segunda función de **Class1.java** tira warnings por posibles parámetros *Integer null*.

2 Ejercicio 2

Para el desarrollo del chequeador de accesos a arrays nos basamos en los visitors del código del Ejercicio 1. En las siguientes secciones detallamos las clases que desarrollamos en el contexto del análisis *dataflow*.

2.1 Reticulado

2.1.1 Intervalos

Definimos un reticulado donde los elementos son vectores $[min, max]$ donde $min \in \mathbb{Z} \cup \{-\infty\}$ y $max \in \mathbb{Z} \cup \{\infty\}$. Para este fin definimos la clase *Interval* que implementa

- operaciones aritméticas entre intervalos ($+, -, \times$; dejamos de lado la división por custiones de tiempo);
- comparaciones básicas de intervalos ($=, \subset$)
- operaciones de conjuntos básicas entre intervalos ($\cup, \cap, \setminus$[1]);
- operaciones que devuelven un intervalo que representa a todos los números menores (resp. mayores) que el intervalo dado;
- una operación auxiliar usada en la función de transferencia, que dados dos intervalos i y j y una relación binaria $\oplus$ ($=, \neq, <, \leq, >, \geq$), provee dos pares de intervalos $(i_t, j_t), (i_f, j_f)$ correspondientes a asumir como verdadera (resp. falsa) la relación binaria $i \oplus j$. Éstos son calculados en base a los intervalos de entrada.

[1] En el caso de $\setminus$ se devuelve, en caso de que el resultado sean dos intervalos disjuntos, uno sólo de ellos

2.1.2 Operaciones de reticulado

Con respecto a las operaciones de reticulado entre intervalos, definimos en la clase *ArrayBoundsLatticeElement*

- como $\top$ al intervalo $[-\infty, \infty]$;
- como $\bot$ al intervalo vacío;
- como $i\ asPreciseAs\ j$ a la relación $i \subseteq j$; y
- como *join* a la unión de dos intervalos en el menor que los contenga a ambos.

2.2 PairLatticeElement

Crystal provee el *mapping TupleLatticeElement*, que utilizamos para relacionar variables con sus intervalos asociados. *PairLatticeElement* es una extensión que incorpora un *mapping* entre variables de array y variables temporales generadas en la representación TAC intermedia.

Mantenemos esta información durante la transferencia para poder acceder en el análisis posterior a la información sobre la longitud de un determinado arreglo que está presente en las variables temporales y no se transfiere a la variable del arreglo original (dado que en el TAC las comparaciones se realizan siempre entre variables temporales).

El *PairLatticeElement* define las mismas operaciones que *ArrayBoundsLatticeElement*, redierccionando a ésta las operaciones, salvo en el caso del join que tiene además un paso extra donde comprueba si es necesario hacer *widening*.

2.2.1 Estrategia de *widening*

Para hacer *widening* mantenemos en un *map* auxiliar un conteo de la cantidad de veces que hicimos *join* del *ForStatement* o *WhileStatement* (subclases de *ASTNode*) dado. Si el conteo supera el máximo arbitrario de 1000 iteraciones, procedemos a mirar qué variables cambiaron en el join y, comparando los límites de los intervalos, los extendemos a $-\infty$ (el mínimo) o a ∞ (al máximo) según corresponda. Esto nos provee convergencia aunque puede dar falsos positivos.

2.3 Función de transferencia

Definimos la función de transferencia como una extensión de la clase de Crystal *AbstractTACBranchSensitiveTransferFunction*, por lo cual nuestro análisis se realiza sobre *Three Address Codes* y es sensible a *branching*. Esta función maneja como elementos del reticulado a objetos del tipo *PairLatticeELement*.

Al comienzo todas las variables del análisis comienzan valiendo $\top$.

Entre las operaciones que realiza tenemos

- copia de intervalos entre variables;
- adquisición del tamaño de un array creado dentro de la función visitada;
- actualización del *mapping* entre *arrays* y variables temporales;
- ejecución abstracta de operaciones aritméticas entre intervalos;

- y resolución de comparaciones entre intervalos (utilizando la auxiliar de *Interval* ya mencionada).

2.4 Análisis de accesos a *array*

Al analizar los resultados del *dataflow* se visita cada acceso a posición de array $arr[i]$ (tanto de lectura como escritura) y se comprueba

- que arr tenga algún intervalo definido distinto a $\bot$; en caso contrario, se informa una advertencia de posible acceso erróneo;
- que el intervalo correspondiente a i no se superponga con el intervalo $[-\infty, -1]$ ni con el intervalo correspondiente a arr; si además de superposición se verifica una contención se puede informar un error en lugar de una advertencia.

Estas reglas se explican porque para cada *array* guardamos el intervalo de sus posibles longitudes; por lo que toda intersección no vacía entre éste y el intervalo de i significa un posible acceso ilegal al *array*. Dado que comenzamos suponiendo que todas las variables son $\top$ ($[-\infty, \infty]$), si no aprendemos nada en el análisis *dataflow* daremos advertencia en cada acceso a array ancontrado (pero no error).

3 Ejercicio 3

Haciendo pruebas sobre **Class2.java** tenemos los siguientes comentarios:

- El análisis detectó como warnings a los accesos en *SumArray1* y *SumArray5*[2].
- A pesar de funcionar como lo esperado, notamos que en *SumArray4* el intervalo calculado para la variable b es $[4, 20]$ en lugar del esperado $[8, 20]$. Notamos que el visitor de Crystal no está procesando correctamente expresiónes del tipo $x = a + b + c + \ldots$ (y es indistinto acá si a, b y c son variables o constantes) dado que sólo termina sumando las dos primeras. Por esto, la expresión $b = a + a + a + a$ termina valiendo $a + a$ lo cual es $[4, 4]$ en ese punto del flujo.
- El *widening* en *SumArray5* fue necesario porque al incrementar la variable a dentro del for no se llegaba al punto fijo. Comentando ese incremento o cambiándolo por algo como $a = i$ nos permitió alcanzar el punto fijo sin necesidad de hacer *widening*.
- Pudimos verificar el funcionamiento del if jugando con los valores de a y b en *SumArray2 SumArray4*. Cambiando a un valor erróneo en sola una de las ramas nos informa una advertencia, y haciéndolo en ambas nos informa un error.

[2]Este falso positivo se debe al widening realizado, aunque lo detectamos como advertencia y no como error.