# Tesis de Licenciatura en Ciencias de la Computación

Descubrimiento automático de restricciones lineales entre variables de programas mediante análisis estático.


Diego José Piemonte - dpiemont@dc.uba.ar - L.U. 638/97

Director: Diego Garbervetsky
Codirector: Victor Braberman

marzo de 2006

A la memoria de José Piemonte.

**Resumen**

Las relaciones entre variables (o invariantes) en un punto del programa son útiles para una gran variedad de tareas como optimización, verificación y especificación de programas, entre otras.

En este trabajo se presenta una técnica de análisis estático de código, y en particular de análisis *Data Flow*, para obtener, en tiempo de compilación, relaciones lineales entre variables en cualquier punto del programa. Dicha técnica está inspirada en un trabajo fundacional presentado a finales de los 70' donde se descubren relaciones lineales para programas "mono-procedurales" que realizan operaciones con números enteros. Durante estos años sólo se presentaron algunos refinamientos o variantes a la técnica original, pero no ha habido ninguna herramienta que pueda lidiar con características de los lenguajes de programación reales y más modernos (procedimientos, orientación a objetos, etc)

En este trabajo se retoma la técnica original y se extiende para analizar programas con procedimientos. Para ello se proponen dos alternativas de solución: la simulación de *inlining* y el análisis simbólico puro (cada procedimiento es analizado de forma aislada y luego vinculado según el contexto de llamada). Otro aporte importante es que en este trabajo se utilizan uniones de poliedros convexos cerrados para representar las restricciones, lo que permite mayor precisión que la que se encuentra en trabajos antecesores.

Este conjunto de técnicas fue implementando en una herramienta que permite evaluar el impacto de las mismas en ejemplos reales. La herramienta permite analizar un subconjunto de Java donde sólo se opera con variables enteras, pero sí se soportan invocaciones a métodos. A nuestro entender, esta es la primer herramienta que permite trabajar con un subconjunto de un lenguaje real.

Finalmente, se realiza una exploración de algunas ideas que podrían posibilitar la extensión de la técnica para analizar programas con características de orientación a objetos. Estas ideas están acompañadas por una extensión a la implementación mencionada anteriormente con la que es posible evaluar el impacto de las mismas. En particular, se pudo descubrir automáticamente relaciones lineales entre variables en un subconjunto considerable de programas del lenguaje Java, donde existen grandes dificultades como *aliasing*, *binding* dinámico, *arrays*, elementos estáticos, entre otros. A pesar de que estas ideas recién se encuentran en gestación y no han sido sustentadas por un marco teórico, creemos que constituyen una buena base para futuros trabajos en pos de una herramienta que permita realizar el análisis en lenguajes modernos como Java.

"La ciencia es orgullosa por lo mucho que ha aprendido;
la sabiduría es humilde por que no sabe más."

# Agradecimientos

La culminación de este trabajo trasciende el mero hecho de la finalización de una tesis de licenciatura, y se transforma en la señal que indica el final de una carrera universitaria que ocupó gran parte de los últimos diez años de mi vida. Es por esto que abusaré de este espacio y, no sólo agradeceré a las personas que estuvieron relacionadas con este trabajo, sino que haré extensivo el agradecimiento a los que me acompañaron durante todo este tiempo.

En primer lugar quiero agradecer a mi amigo y director Diego Garbervetsky por su descomunal dedicación en la dirección de esta tesis, que significó para mí una gran ayuda en un momento muy difícil. A Victor Braberman por aportar toda su experiencia y conocimientos en la codirección. A Dependex por brindarme los recursos necesarios para la realización de este trabajo. Alan, Andran, Esteban, Fer$^2$, Guido, Juan, Lito y Lu son responsables en la generación del excelente clima de trabajo en el que se desarrolló esta tesis.

A Nicolás Kicillof quien desde un primer momento estuvo consustanciado con esta tesis, batió el record en la relación $\frac{minuciosidad}{tiempo}$ de su lectura y realizó excelentes sugerencias para mejorarla. A Marcelo Scasso quien me acompaña desde hace varios años en muchas facetas de mi vida, por todo lo que hizo siempre por mí y porque es un ídolo.

A Fidel quien me enseño a dar los primeros pasos en el mundo de la investigación, y a quien le debo mucho de lo que aprendí.

A todos los docentes que tuve a lo largo de la carrera entre los que me gustaría destacar a Mónica Bobrowski quien me dio la posibilidad de ejercer esa maravillosa actividad que es la docencia. A Eduardo Fermé quien tuvo la habilidad de enseñarme mucho y divertirme mucho al mismo tiempo. A Emilio Platzer por la pasión con la que da sus clases.

A mis compañeros de carrera por todo lo que eso significó: Ana, Danis B. y T., Diego, Javi, Juan R., Lau, León, Mati, Maxi, Pablos Be. y Bu., Pato, Pecho, Riki, Romi, Tincho.

A los "borrachos" del DC: Dani, Demián, Die, Esteban, Greg, JP, Pablic, Santi y Sergios M. y D. por su grata compañía.

A Magda por prestarme a Diego unos ratitos.

A NT, y especialmente a Esteban, por brindarme la posibilidad de que mi actividad académica y mi actividad en la industria convivan en armonía.

A Cris, Flac y Martín$^2$ por el Rock & Roll que supimos conseguir...

A todos mis amigos. Los más viejos: Augus, Ale, Luqui, Pablo. Los del oeste: Bolo, Cabra, Maxi, Oso, Raqui. Los de El Colegio y los tercerizados: Ana$^2$, Dani, Esteban,

Ire, Luciano, Shunko, Pablo. Los “tallereantes” y “crmeros”: Agus D. y B., Ales S. y F., Bajtul, Belén, Caver, Charly, Doctor, Ladri, Leo, Lu, Martín, Oscar. Los transoceánicos: Ali, Bianca, Borja, Graeme, Juan, Matt, Mónica, Saúl, Rolando, Roman.

A mi familia por la increíble contención que me brindan. En primer lugar a mi mamá a quien le debo estar ahora escribiendo esto, lo digo no sólo por la habitual referencia a haberme dado la vida, sino también por la educación; además de todo su cariño y fuerza en los momentos difíciles. A Nachi quien siempre me acompaña en todo, aun cuando no estamos juntos. A la Abu que es genial. A mis tías y tíos que me quieren mucho: Ana, Narda, Silvia, Olga y Vic[2]; a mis primos con quienes paso muy gratos momentos: Augus, Ana, Edu, Ena, Emi, Fab, Juan y Lean. Y a Norma, una madre más.

A los que ya no están y no pueden leer esto, quienes igualmente están en mi corazón.

Y a Vero, a quien le debo mi felicidad, por su paciencia y fortaleza en estos últimos tiempos, y por su infinito y velozmente divergente amor.

# Índice general

# Capítulo 1

# Introducción

Las restricciones entre variables de un programa son predicados que permiten determinar cuáles son los potenciales valores que dichas variables podrán adoptar cuando el programa es ejecutado. Existen varios contextos en los que es posible considerar las mencionadas restricciones: los invariantes de representación que predican sobre la validez de una instancia particular de una estructura de datos, y los *object invariants* [CL05b] que describen propiedades comunes a todas las instancias de determinada clase, son algunos ejemplos de contextos en los que se consideran las restricciones entre variables. El contexto considerado en este trabajo es el *punto del programa*, es decir, se determinarán restricciones entre variables antes y después de la ejecución de cada instrucción (o punto) del programa. Estas restricciones están estrechamente ligadas al *program counter* y, en consecuencia, se dice que son locales.

En la figura 1.1 se presenta un pequeño programa y restricciones entre variables que fueron calculadas "a mano" a partir del código fuente del programa, para el instante previo y posterior de la ejecución de cada sentencia. Es posible ver que la restricción entre variables antes de la sentencia "`1:`" está representada por el predicado universalmente válido { `TRUE` }. Esto significa que, para este punto del programa, no se considera ninguna restricción y que las variables podrían adoptar cualquier valor en tiempo de ejecución[1]. La restricción presente después de la sentencia "`1:`" indica que cuando el programa sea ejecutado el valor de la variable `a` será 3 para cualquier posible ejecución del programa (cualquier combinación de parámetros de entrada). Por otro lado, se observa que en la restricción existente antes de la sentencia "`4:`" se considera el hecho de que se está dentro de una rama del `if`, y la restricción antes de la sentencia "`6:`" considera la unión de ambas ramas del `if`. Cabe aclarar que en la restricción existente después de la sentencia "`5:`" la relación $c = a + 1$ ya no es válida porque la sentencia "`5:`" modifica el valor de la variable $a$; es por esto que la relación correspondiente debe ser $c = 4$.

**Definición 1.0.1 (Restricción de precisión absoluta)** *Se dice que una restricción para un punto del programa es de* precisión absoluta *cuando existe, al menos, una posible ejecución del programa (combinación de valores de los argumentos) en la cual los potenciales valores de las variables que describe la restricción son efectivamente tomados por*

[1] Luego se verá que resulta mucho más conveniente considerar como restricción existente antes de la primera instrucción del programa a aquella que representa el *binding* entre los parámetros formales y meta-variables que representan los argumentos con los cuales se realiza una invocación genérica. En el caso de la figura 1.1 sería $\{\ p1 = p1_0 \wedge p2 = p2_0\ \}$

```
    void programa1(int p1, int p2)
    {
        int a, b, c, d;
1:      a = 3;
2:      b = p1 ^ 2;
3:      c = a + 1;
        if (p1 >= p2)
4:          d = p1 * p2;
        else
5:          a = b;
6:      return;
    }
```

| | | |
|---|---|---|
| 1: | antes | { TRUE } |
| | después | $\{ a = 3 \}$ |
| 2: | antes | $\{ a = 3 \}$ |
| | después | $\{ a = 3 \wedge b = p1^2 \}$ |
| 3: | antes | $\{ a = 3 \wedge b = p1^2 \}$ |
| | después | $\{ a = 3 \wedge b = p1^2 \wedge c = a + 1 \}$ |
| 4: | antes | $\{ a = 3 \wedge b = p1^2 \wedge c = a + 1 \wedge p1 \geq p2 \}$ |
| | después | $\{ a = 3 \wedge b = p1^2 \wedge c = a + 1 \wedge p1 \geq p2 \wedge d = p1 * p2 \}$ |
| 5: | antes | $\{ a = 3 \wedge b = p1^2 \wedge c = a + 1 \wedge p1 < p2 \}$ |
| | después | $\{ a = b \wedge b = p1^2 \wedge c = 4 \wedge p1 < p2 \}$ |
| 6: | antes | $\{ a = 3 \wedge b = p1^2 \wedge c = a + 1 \wedge p1 \geq p2 \wedge d = p1 * p2 \} \cup$ $\{ a = b \wedge b = p1^2 \wedge c = 4 \wedge p1 < p2 \}$ |
| | después | $\{ a = 3 \wedge b = p1^2 \wedge c = a + 1 \wedge p1 \geq p2 \wedge d = p1 * p2 \} \cup$ $\{ a = b \wedge b = p1^2 \wedge c = 4 \wedge p1 < p2 \}$ |

Figura 1.1: Restricciones entre variables calculadas "a mano" para un programa simple.

*las variables del programa. O sea, la restricción describe exactamente el conjunto de los valores posibles que pueden adoptar las variables en el punto del programa correspondiente, considerando todas las trazas existentes de toda posible combinación de valores de los argumentos.*

Por ejemplo, todas las restricciones de la figura 1.1 son de precisión absoluta. Sin embargo, si se considera como restricción después de la sentencia "`1:`" a $\{ a = 3 \} \cup \{ a = 4 \}$, esta restricción no es de precisión absoluta ya que, si bien es correcta en el sentido en que describe cuáles son los potenciales valores de la variable `a` cuando el programa es ejecutado, no existe en la práctica ninguna ejecución posible en la cual el valor de la variable `a` sea 4.

**Observación 1.1** *Las restricciones existentes en la figura 1.1 están compuestas tanto de relaciones lineales entre variables, por ejemplo* $a = 3$ *o* $p1 \geq p2$*, como de relaciones no lineales, por ejemplo* $b = p1^2$ *o* $d = p1 * p2$*.*

## 1.1. Precisión y computabilidad

Como se mencionó anteriormente, las restricciones entre variables presentes en la figura 1.1 son de precisión absoluta y fueron calculadas "a mano" a partir del código fuente del programa. Con esta misma técnica es posible obtener las restricciones de precisión absoluta de la figura 1.2. Cabe aclarar que sería un error considerar que la

restricción antes de la sentencia "`3:`" es $\{\ a = 3 \wedge b = 0\ \}$, ya que la misma sería válida únicamente para la primera iteración del ciclo, dado que para las siguientes iteraciones la relación $b = 0$ no se mantiene.

```
        int programa2()
        {
            int a, b;
1:          a = 3;
2:          b = 0;
            while ( a == 3)
            {
3:              b = b + 1;
            }
4:          return b;
        }
```

| | | |
|---|---|---|
| 1: | antes | { `TRUE` } |
| | después | $\{\ a = 3\ \}$ |
| 2: | antes | $\{\ a = 3\ \}$ |
| | después | $\{\ a = 3 \wedge b = 0\ \}$ |
| 3: | antes | $\{\ a = 3 \wedge b \geq 0\ \}$ |
| | después | $\{\ a = 3 \wedge b > 0\ \}$ |
| 4: | antes | { `FALSE` } |
| | después | { `FALSE` } |

Figura 1.2: Restricciones entre variables calculadas "a mano" para un programa que no se detiene.

La restricción existente antes de la sentencia "`4:`" de la figura 1.2 es { `FALSE` }, o sea el predicado que representa una contradicción. Esta restricción indica que, cuando el programa sea ejecutado, no existe ninguna combinación de valores posibles para las variables `a` y `b` en el instante previo a la ejecución de dicho punto del programa. De hecho, una rápida observación del programa en cuestión permite constatar que la condición de corte del ciclo nunca será alcanzada, por lo tanto si la restricción antes de la la sentencia "`4:`" diera la posibilidad a alguna variable de tomar algún valor, la restricción no sería de precisión absoluta ya que ninguna ejecución del programa haría que la variable obtenga dicho valor. Esta situación significa que el flujo del programa nunca podrá atravesar la sentencia "`4:`", es decir la sentencia "`4:`" es en realidad "código muerto". Luego, teniendo en cuenta que el único punto de retorno del programa es "código muerto" podemos afirmar que el programa, al ser ejecutado, no se detendrá.

Teniendo en cuenta, por un lado, que el objetivo de esta tesis es el descubrimiento *automático* de restricciones, es decir, evitar el cálculo "a mano" y delegar dicha tarea a una computadora. Y, por otro lado, teniendo en cuenta que existen casos como el presente en el ejemplo de la figura 1.2, es necesario detenerse a analizar las expectativas que deben tenerse antes de intentar desarrollar un algoritmo que resuelva el problema.

**Proposición 1.1.1** *El cálculo de restricciones de precisión absoluta no es computable.*

**Demostración:** Se supone que el cálculo de restricciones de precisión absoluta es computable. Entonces existe el programa $P$ que recibe un programa $x$ y calcula las restricciones de precisión absoluta correspondientes. Ahora se construye un programa $Q$ que recibe como argumento un programa $x$, aplica el programa $P$ a $x$ $(P(x))$, y devuelve

*true* si existe al menos un punto de retorno[2] del programa $x$ para el cual la restricción calculada antes de dicho punto es diferente de { `FALSE` }, y devuelve *false* en caso de que las restricciones existentes antes de cada punto de retorno sean todas { `FALSE` }. Luego el programa $Q$ es un programa que recibe un programa $x$ e informa si existe alguna posible ejecución de $x$ que se detiene. El programa $Q$ es una versión del *halting problem* que se demuestra no computable, por lo tanto el programa $Q$ no existe y se obtiene un absurdo por suponer que el programa $P$ existe. □

La proposición 1.1.1 es útil para saber que no será posible construir un algoritmo que calcule restricciones de precisión absoluta para todo programa dado. Sin embargo, el problema de calcular restricciones no necesariamente de precisión absoluta es computable y se resuelve trivialmente mediante un algoritmo que calcula siempre la restricción { `TRUE` } antes y después de cada instrucción del programa. Claramente este último algoritmo carece totalmente de utilidad. Pero afortunadamente existe un rango de posibilidades entre el problema no computable de calcular restricciones de precisión absoluta y la solución trivial al cálculo de restricciones no necesariamente de precisión absoluta.

La técnica presentada en este trabajo toma el código fuente de un programa y calcula, sin necesidad de ejecutarlo, restricciones de precisión no necesariamente absoluta, compuestas únicamente de relaciones lineales (ver observación 1.1), de donde se cumplen las siguientes relaciones de fuerza:

$$\textit{Restricción de precisión absoluta} \Rightarrow \textit{Restricción computada} \Rightarrow \{ \texttt{TRUE} \}$$

Además, se espera que en la gran mayoría de los casos se cumpla:

$$\textit{Restricción computada} \neq \{ \texttt{TRUE} \}$$

A partir de la relación:

$$\textit{Restricción de precisión absoluta} \Rightarrow \textit{Restricción computada}$$

se dice que la técnica que se presenta en este trabajo calcula *aproximaciones conservadoras.*

## 1.2. Motivación

Ya se ha visto en la sección 1.1 que no es posible obtener siempre automáticamente restricciones de precisión absoluta. Sin embargo, como se muestra en este trabajo, sí es posible obtener "buenas"[3] aproximaciones conservadoras partiendo únicamente del código fuente del programa (sin ejecutarlo). Contar con restricciones lineales no necesariamente de precisión absoluta, obtenidas a partir del código fuente del programa, permite conocer en tiempo de compilación rangos, tanto numéricos como simbólicos, dentro de los cuales

[2] Un punto de retorno de un programa es una sentencia de la forma `return;` o `return expr;`

[3] Si bien el concepto de "buena" aproximación no ha sido formalizado, se entenderá que una aproximación es "buena" cuando se encuentre cercana a la restricción de precisión absoluta en una hipotética secuencia (o grafo dirigido acíclico) donde todas las restricciones válidas son ordenadas mediante la relación de fuerza.

se encontrarán los valores de las expresiones del programa en tiempo de ejecución. Esto es muy útil para una gran cantidad de tareas que se describen a continuación.

### 1.2.1. Optimización

Las restricciones lineales pueden ser utilizadas para realizar propagación de constantes. Por ejemplo, si se considera la restricción antes de la sentencia "`3:`" de la figura 1.1, se puede observar que para todas las ejecuciones posibles del programa la variable `a` siempre adoptará en valor 3, por lo tanto en la sentencia "`3:`" es posible reemplazar directamente en el código del programa a la variable `a` por su valor, dando lugar a la sentencia `c = 3 + 1`. Luego será posible reducir la expresión a c = 4 y eventualmente volver a reemplazar el código del programa, donde la operación aritmética es eliminada.

Otra aplicación a la optimización es la eliminación de *código muerto* [CH78]. Como se vio en la sección 1.1, en algunas circunstancias será posible determinar que una instrucción es *código muerto*, lo que permitirá reescribir el programa obteniendo una versión más pequeña.

### 1.2.2. Verificación

Existen varias verificaciones que pueden hacerse utilizando restricciones lineales entre variables. Una de ellas es la no terminación de un programa (ver sección 1.1).

Otra verificación posible es la del acceso a *arrays* dentro de los límites permitidos. Esta tarea se debe realizar teniendo en cuenta que las restricciones obtenidas son aproximaciones. Por ejemplo, si se accede a un *array* de dimensión 10 utilizando como índice la variable `i` y sabiendo que se tiene la siguiente restricción $\{ i \geq 0 \wedge i \leq 11 \}$, no debe reportarse un error, sino una "advertencia" de que existe la posibilidad de que se realice un acceso fuera de los límites del *array*. Si por el contrario la restricción obtenida, en el mismo punto que el ejemplo anterior, es $\{ i = 11 \}$, sí debe reportarse un "error" ya que fue posible determinar en tiempo de compilación que el programa accederá indefectiblemente fuera de los límites del *array* en cuestión. De esta misma forma también es posible verificar la creación de *arrays* utilizando como dimensión un número no negativo, condición necesaria en algunos lenguajes de programación, como por ejemplo Java.

Si un programa realiza invocaciones a determinada función y se cuenta con un predicado especificando la precondición de dicha función es posible verificar, en tiempo de compilación, para cada punto de invocación, si la precondición se cumplirá para todas las posibles ejecuciones del programa, si no es posible asegurar que la precondición se cumpla para todas las ejecuciones o si es posible asegurar que la precondición no se cumplirá en ninguna ejecución posible. Por ejemplo, si consideramos el programa de la figura 1.3, podemos observar que hay tres invocaciones a la función `factorial(int param)`, cuya post-condición se asume representada por la restricción $\{ retValue = param! \}$.

Suponiendo que se cuenta con un predicado que describe la precondición de la función `factorial(int param)`, por ejemplo $\{ param \geq 0 \}$, se pueden realizar los siguientes razonamientos. La primera invocación, sentencia "`1:`", tiene como restricción antes de la ejecución a $\{ i = 5 \}$, por lo tanto:

$$\{p = p_0 \wedge i = 5\} \wedge \{i = param\} \Rightarrow \{param \geq 0\}$$

```
    void programa3(int p)
    {
        int i, a, b, c;
        i = 5;
1:      a = factorial(i);
2:      b = factorial(p);
        while ( i >= 0)
        {
            i--;
        }
3:      c = factorial(i);
        return;
    }
```

| | | |
|---|---|---|
| 1: | antes | $\{\ p = p_0 \wedge i = 5\ \}$ |
| | después | $\{\ p = p_0 \wedge i = 5 \wedge a = i!\ \}$ |
| 2: | antes | $\{\ p = p_0 \wedge i = 5 \wedge a = i!\ \}$ |
| | después | $\{\ p = p_0 \wedge i = 5 \wedge a = i! \wedge b = p!\ \}$ |
| 3: | antes | $\{\ p = p_0 \wedge i = -1 \wedge a = 5! \wedge b = p!\ \}$ |
| | después | $\{\ p = p_0 \wedge i = -1 \wedge a = 5! \wedge b = p! \wedge c = i!\ \}$ |

Figura 1.3: Restricciones calculadas "a mano" para un programa que realiza invocaciones a la función $factorial$.

En consecuencia la primera invocación siempre se realizará respetando la precondición.

De la misma forma, para la segunda invocación se tiene que:

$$\{p = p_0 \wedge i = 5 \wedge a = i!\} \wedge \{p_0 = param\} \not\Rightarrow \{param \geq 0\} \wedge$$

$$\{p = p_0 \wedge i = 5 \wedge a = i!\} \wedge \{p_0 = param\} \not\Rightarrow \neg\{param \geq 0\}$$

En consecuencia no es posible determinar nada en tiempo de compilación sobre la invocación de la sentencia "`2:`"

Por último, para la tercera invocación se tiene que:

$$\{p = p_0 \wedge i = -1 \wedge a = 5! \wedge b = p!\} \wedge \{i = param\} \Rightarrow \neg\{param \geq 0\}$$

lo que permite concluir que nunca se cumplirá la precondición de la función en ese punto del programa.

En conclusión, es posible decir que estas mismas ideas son aplicables a la verificación en tiempo de compilación de contratos o aserciones.

### 1.2.3. Especificación

Las restricciones entre variables pueden ser utilizadas tanto para la obtención de especificaciones, como la verificación de la correspondencia de un programa con una especificación.

Con respecto a la obtención de especificaciones es posible distinguir dos casos: obtención de post-condiciones y obtención de pre-condiciones. Para el primer caso se debe

realizar la disyunción de las restricciones obtenidas después de cada punto de retorno del programa. Por ejemplo, si se considera el programa de la figura 1.4:

```
        int maximo(int a, int b)
        {
            if (a >= b)
            {
1:              return a;
            }
            else
            {
2:              return b;
            }
        }
```

| | | |
|---|---|---|
| 1: | antes | $\{\ a = a_0 \wedge b = b_0 \wedge a \geq b\ \}$ |
| | después | $\{\ a = a_0 \wedge b = b_0 \wedge a \geq b \wedge retValue = a_0\ \}$ |
| 2: | antes | $\{\ a = a_0 \wedge b = b_0 \wedge a < b\ \}$ |
| | después | $\{\ a = a_0 \wedge b = b_0 \wedge a < b \wedge retValue = b_0\ \}$ |

Figura 1.4: Obtención de post-condiciones

se puede observar que existen dos puntos de retorno del programa. Además cada punto de retorno tiene el efecto de ligar la expresión retornada con una meta-variable introducida para representar el valor devuelto por el programa. En esta caso la post-condición obtenida es:

$$\{a = a_0 \wedge b = b_0 \wedge a \geq b \wedge retValue = a_0\} \vee \{a = a_0 \wedge b = b_0 \wedge a < b \wedge retValue = b_0\}$$

Es posible observar cómo el predicado anterior refleja exactamente la post-condición de la función. Sin embargo; teniendo en cuenta que las restricciones que se obtendrán automáticamente son, en general, aproximaciones; se presenta el caso en el cual la post-condición real de la función implica lógicamente a la post-condición obtenida automáticamente en tiempo de compilación, es decir las post-condiciones obtenidas automáticamente pueden ser, en general, más débiles.

Para el caso de la obtención automática de precondiciones es necesario ligar, antes de la primera sentencia del programa, los parámetros formales con meta-variables que representarán valores genéricos para cualquier invocación. Luego analizando las post-condiciones obtenidas es posible obtener información sobre la pre-condición. Por ejemplo, si se considera el programa de la figura 1.5, se observa que antes del único punto de retorno del programa se cumple la restricción $\{\ a_0 \leq b_0\ \}$. Esto implica que en todas las posibles ejecuciones en las cuales el programa termina se cumple que $\{\ a_0 \leq b_0\ \}$, y por lo tanto se puede inferir la precondición $\{\ a \leq b\ \}$.

En lo que respecta a verificar si un programa cumple o no determinada especificación, pueden realizarse las siguientes tareas. Si se cuenta con una post-condición del programa es posible calcular automáticamente la post-condición de la forma mencionada anteriormente, y luego verificar que:

$$\textit{postcondición a verificar} \not\Rightarrow \neg \textit{poscondición calculada automáticamente}$$

```
    void programa4(int a, int b)
    {
        if (a >= b)
        {
            while (a > b)
            {
                a++;
            }
        }
1:      return;
    }
```

| | | |
|---|---|---|
| 1: | antes | $\{\ a = a_0 \wedge b = b_0 \wedge a_0 \leq b_0\ \}$ |
| | después | $\{\ a = a_0 \wedge b = b_0 \wedge a_0 \leq b_0\ \}$ |

Figura 1.5: Obtención de precondiciones

de no cumplir dicha condición, o sea en caso que:

$$\textit{postcondición a verificar} \Rightarrow \neg \textit{poscondición calculada automáticamente}$$

Se puede afirmar que el programa no respeta la especificación dada. Notar que debido a que las post-condiciones calculadas automáticamente son aproximaciones si se presenta la siguiente situación:

$$\textit{postcondición a verificar} \not\Rightarrow \textit{poscondición calculada automáticamente}$$

no es posible afirmar que el programa no cumple con la post-condición especificada, sino que simplemente la post-condición calculada no es lo suficientemente precisa.

### 1.2.4. Técnicas adicionales

Adicionalmente a las motivaciones clásicas descriptas anteriormente, existen algunas técnicas que se basan en la información que pueden proveerle las restricciones entre variables para un punto del programa. Por ejemplo, la técnica presentada en [BGY05] se basa en la información provista por las restricciones entre variables para sobre-aproximar mediante una expresión simbólica el consumo de memoria de un programa. Por otro lado, en [TS05], se presenta una técnica para la obtención automática de casos de prueba que cubren todos las posibles alternativas de flujo.

## 1.3. Trabajos relacionados

Un punto de partida en el descubrimiento automático de restricciones lineales entre variables de programas puede situarse en [CH78]. En este trabajo se presenta una técnica para encontrar restricciones lineales utilizando interpretación abstracta [CC77]. El dominio abstracto utilizado es el de poliedros convexos, o sea, las restricciones lineales descubiertas se representan mediante poliedros convexos. La técnica desarrollada en [CH78] es *intra-procedural*, es decir, sólo puede ser aplicada a un procedimiento (o

función) sin que haya llamados a procedimientos (o funciones) auxiliares. Además, esta técnica puede ser aplicada a programas con un control de flujo simple, sin soportar características de lenguajes orientados a objetos.

En [CC02] se presenta una serie de métodos para realizar análisis estático modular de programas. Si bien no se presenta explícitamente como una extensión de la técnica de [CH78] para soportar análisis modular (llamados a procedimientos o funciones), puede considerarse que sienta las bases para realizar dicha extensión.

En lo que respecta a dominios abstractos, en [FR99] se presenta el operador *powerset* que, aplicado a un dominio abstracto base, permite obtener abstracciones más precisas. Combinando estas ideas con las presentadas en [BHZ04] es posible incrementar la precisión de la técnica presentada en [CH78].

En [Min00] se presenta una técnica muy similar a la de este trabajo, pero con un enfoque diferente. El dominio abstracto que se utiliza en aquél es *Two-variable difference or sum constraint sets*, lo que permite órdenes de complejidad algorítmica mucho menores en detrimento de la precisión de los resultados obtenidos. Además, la técnica se desarrolla sobre un lenguaje imperativo muy simple, sin existir análisis inter-procedural ni características de lenguajes orientados a objetos.

En [SSM04] se presenta una técnica para descubrir invariantes lineales en sistemas de transición. Estos sistemas son una abstracción del programa, en la que las bifurcaciones de flujo son consideradas como no determinísticas. Las restricciones se descubren utilizando un enfoque completamente diferente al que se encuentra en [CH78] llamado *Constraint-based technique*. La técnica se declara como más escalable y, en algunas ocasiones, más precisa que [CH78]. Sin embargo no es interprocedural y no se contemplan características de lenguajes orientados a objetos.

Al efectuar análisis estático sobre programas escritos en lenguajes orientados a objetos, se hace necesario realizar algún tipo de *points-to analysis*. De esta manera es posible predecir conservativamente las posiciones de memoria referenciadas por un puntero.En el segundo capítulo de [Lho02] es posible encontrar una descripción bastante precisa de la bibliografía más representativa del area, junto a una descripción de su evolución hasta alcanzar lenguajes modernos como, por ejemplo, Java.

Existen determinados programas cuya ejecución puede hacer crecer el *heap* a tamaños arbitrariamente grandes. Al realizar análisis estático sobre dichos programas se hace necesario obtener abstracciones manipulables de dicho *heap*. *Shape Analysis* es el área de estudio encargada de obtener esas abstracciones. Existe una basta bibliografía relacionada, entre la que es posible destacar a [SRW99].

La herramienta presentada en [ECGN00] permite obtener predicados que describan los potenciales valores de las variables para programas escritos en lenguajes con características de orientación a objetos, como Java. Sin embargo la técnica tiene varios inconvenientes. Primero, no permite obtener dicha información en tiempo de compilación, sino que por el contrario se basa en la ejecución sistemática del programa. En segunda instancia, dado que el resultado que brinda la herramienta se basa en ejecuciones, el mismo está estrechamente ligado a los datos de entrada, pudiendo incluso dar soluciones erróneas si los datos de entrada utilizados no son adecuados. Un tercer inconveniente es que sólo brinda información sobre precondiciones y post-condiciones de los métodos ejecutados (pseudo-contratos), no siendo posible obtener resultados para cualquier punto del programa.

## 1.4. Estructura de la tesis

En el capítulo 2 se presenta una serie de conceptos y definiciones preliminares que serán utilizados frecuentemente en el desarrollo de las técnicas en los siguientes capítulos.

En el capítulo 3 se presentan las técnicas principales de la tesis en las que se extiende el trabajo presentado en [CH78]. El principal aporte es el análisis de programas con llamados a procedimientos utilizando dos estrategias diferentes: *inlining* y análisis simbólico. Además, se utilizan uniones de poliedros convexos cerrados para representar las restricciones, permitiendo mayor precisión en las restricciones obtenidas que la que se encuentra en [CH78].

En el capítulo 4 se presentan las principales características de la herramienta que implementa las técnicas del capítulo 3. Además, se muestran y analizan algunos resultados obtenidos con la implementación.

En el capítulo 5 se discuten y se plantean posibles extensiones a las ideas presentadas en el capítulo 3 con el fin de soportar características habituales de los lenguajes orientados a objetos tales como *aliasing*, *binding* dinámico, *arrays*, elementos estáticos, entre otros. Las ideas de este capítulo son los primeros acercamientos a un enfoque global para soportar estos lenguajes, por lo tanto no se realiza un desarrollo formal. Sin embargo sí se presenta una implementación de características de "*proof of concept*" que se utiliza para determinar el potencial, la aplicabilidad y el impacto de las ideas mencionadas.

En el capítulo 6 se presentan las conclusiones de esta tesis y se discuten algunas alternativas de trabajo a futuro.

# Capítulo 2

# Definiciones preliminares

En este capítulo se presenta una serie de conceptos sobre los que se basa la técnica desarrollada en este trabajo.

## 2.1. Dominio de poliedros convexos cerrados

Si se vuelve a considerar la figura 1.2 y se analizan, por ejemplo, las restricciones existentes después la sentencia "`2:`" y antes de la sentencia "`3:`", es posible observar que existe una interpretación geométrica para estas restricciones. Es decir, la restricción existente después la sentencia "`2:`" ($\{\ a = 3 \wedge b = 0\ \}$) representa el punto del plano $(3, 0)$, y la restricción antes de la sentencia "`3:`" ($\{\ a = 3 \wedge b \geq 0\ \}$) representa la recta definida por el conjunto de puntos $\{(x, y) | x = 3 \wedge y \geq 0\}$ tomando $a$ como las abscisas y $b$ las ordenadas.

Partiendo de esta idea, es posible representar las restricciones entre variables utilizando figuras geométricas $n$-dimensionales, o sea conjuntos de puntos en un espacio $n$-dimensional. En este trabajo, tal como se describe en [CH78], se considerará como elemento principal para describir restricciones entre variables, a los conjuntos de puntos que describen los poliedros convexos cerrados.

**Definición 2.1.1 (Poliedro convexo cerrado)** *Un conjunto de puntos $P$ de $\mathbb{Z}^n$, es un poliedro convexo cerrado de dimensión $n$ si:*

$$P = \{x | Ax \leq b\}$$

*para una matriz $A$ de dimensión $m \times n$ y un vector $b$ de dimensión $m$. Donde $m$ es la cantidad de restricciones lineales que se intersecan para formar el poliedro de dimensión $n$.*

A continuación se muestra como representar relaciones genéricas entre variables

- Para representar una restricción de la forma:

$$\sum_{i=1}^{n} k_i x_i \leq c$$

se debe considerar como matriz y vector a:

$$A = \begin{pmatrix} k_1 & k_2 & ... & k_n \end{pmatrix}$$

$$b = \begin{pmatrix} c \end{pmatrix}$$

- Para representar una restricción de la forma:

$$\sum_{i=1}^{n} k_i x_i < c$$

se debe considerar como matriz y vector a:

$$A = \begin{pmatrix} k_1 & k_2 & ... & k_n \end{pmatrix}$$

$$b = \begin{pmatrix} c - 1 \end{pmatrix}$$

- Para representar una restricción de la forma:

$$\sum_{i=1}^{n} k_i x_i \geq c$$

se debe considerar como matriz y vector a:

$$A = \begin{pmatrix} -k_1 & -k_2 & ... & -k_n \end{pmatrix}$$

$$b = \begin{pmatrix} -c \end{pmatrix}$$

- Para representar una restricción de la forma:

$$\sum_{i=1}^{n} k_i x_i > c$$

se debe considerar como matriz y vector a:

$$A = \begin{pmatrix} -k_1 & -k_2 & ... & -k_n \end{pmatrix}$$

$$b = \begin{pmatrix} -c - 1 \end{pmatrix}$$

- Finalmente, para representar una restricción de la forma:

$$\sum_{i=1}^{n} k_i x_i = c$$

se debe considerar como matriz y vector a:

$$A = \begin{pmatrix} k_1 & k_2 & ... & k_n \\ -k_1 & -k_2 & ... & -k_n \end{pmatrix}$$

$$b = \begin{pmatrix} c \\ -c \end{pmatrix}$$

En particular suponiendo que se tiene el conjunto de variables $\{x_1, x_2, ..., x_n\}$ se pueden representar las siguientes expresiones de la forma indicada:

- la restricción: $x_i = c$ se representa mediante el poliedro que se obtiene considerando como matriz y vector a:

$$A = \begin{pmatrix} 0_1 & 0_2 & ... & 0_{i-1} & 1 & 0_{i+1} & ... & 0_n \\ 0_1 & 0_2 & ... & 0_{i-1} & -1 & 0_{i+1} & ... & 0_n \end{pmatrix}$$

$$b = \begin{pmatrix} c \\ -c \end{pmatrix}$$

- la restricción: $x_i = x_j$ con $i \neq j$ se representa mediante el poliedro que se obtiene considerando como matriz y vector a:

$$A = \begin{pmatrix} 0_1 & 0_2 & ... & 0_{i-1} & 1 & 0_{i+1} & ... & 0_{j-1} & -1 & 0_{j+1} & ... & 0_n \\ 0_1 & 0_2 & ... & 0_{i-1} & -1 & 0_{i+1} & ... & 0_{j-1} & 1 & 0_{j+1} & ... & 0_n \end{pmatrix}$$

$$b = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

- la restricción: $x_i = cx_j$ con $i \neq j$ se representa mediante el poliedro que se obtiene considerando como matriz y vector a:

$$A = \begin{pmatrix} 0_1 & 0_2 & ... & 0_{i-1} & 1 & 0_{i+1} & ... & 0_{j-1} & -c & 0_{j+1} & ... & 0_n \\ 0_1 & 0_2 & ... & 0_{i-1} & -1 & 0_{i+1} & ... & 0_{j-1} & c & 0_{j+1} & ... & 0_n \end{pmatrix}$$

$$b = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

- la restricción: $x_i = x_j + c$ con $i \neq j$ se representa mediante el poliedro que se obtiene considerando como matriz y vector a:

$$A = \begin{pmatrix} 0_1 & 0_2 & ... & 0_{i-1} & 1 & 0_{i+1} & ... & 0_{j-1} & -1 & 0_{j+1} & ... & 0_n \\ 0_1 & 0_2 & ... & 0_{i-1} & -1 & 0_{i+1} & ... & 0_{j-1} & 1 & 0_{j+1} & ... & 0_n \end{pmatrix}$$

$$b = \begin{pmatrix} c \\ -c \end{pmatrix}$$

- la restricción: $x_i + x_j = x_k$ con $i \neq j \wedge j \neq k \wedge i \neq k$ se representa mediante el poliedro que se obtiene considerando como matriz y vector a:

$$A = \begin{pmatrix} 0_1 & ... & 0_{i-1} & 1 & 0_{i+1} & ... & 0_{j-1} & 1 & 0_{j+1} & ... & 0_{k-1} & -1 & 0_{k+1} & ... & 0_n \\ 0_1 & ... & 0_{i-1} & -1 & 0_{i+1} & ... & 0_{j-1} & -1 & 0_{j+1} & ... & 0_{k-1} & 1 & 0_{k+1} & ... & 0_n \end{pmatrix}$$

$$b = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Si se quiere que una restricción sea la conjunción de más de una relación entre variables simplemente se construye una matriz en la cual se concatenan las filas de las matrices correspondientes a cada relación incluida en la restricción; y también se concatenan los vectores respetando el orden utilizado para la concatenación de filas de matrices.

Por ejemplo, la restricción $\{ x_1 = 3 \wedge x_2 = 0 \}$ se representa mediante el poliedro que se obtiene considerando:

$$A = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix}$$

$$b = \begin{pmatrix} 3 \\ -3 \\ 0 \\ 0 \end{pmatrix}$$

La matriz $A$ se obtiene concatenando las matrices:

$$A' = \begin{pmatrix} 1 & 0 \\ -1 & 0 \end{pmatrix}$$

$$A'' = \begin{pmatrix} 0 & 1 \\ 0 & -1 \end{pmatrix}$$

El vector $b$ se obtiene concatenando los vectores:

$$b' = \begin{pmatrix} 3 \\ -3 \end{pmatrix}$$

$$b'' = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Adicionalmente, la restricción $\{ x_1 = 3 \wedge x_2 \geq 0 \}$ se representa mediante el poliedro que se obtiene considerando:

$$A = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$b = \begin{pmatrix} 3 \\ -3 \\ 0 \end{pmatrix}$$

Cabe destacar, que la razón por la cual las restricciones que son descubiertas por la técnica presentada en este trabajo están compuestas únicamente por componentes lineales (ver observación 1.1) es, precisamente, que la abstracción utilizada para representar dichas restricciones está basada en poliedros convexos cerrados, y los mismos no pueden ser utilizados para representar relaciones no lineales. Otro aspecto a considerar es que los poliedros que son tenidos en cuenta son *cerrados*, o sea los determinados por la relación "$\leq$" y no por la relación "$<$". Esto se debe a que el análisis se lleva a cabo en programas que realizan operaciones con números enteros, por lo tanto no es necesaria la relación "$<$" que sí lo sería, por ejemplo, en presencia de conjuntos de elementos densos como los reales.

Si se analiza la capacidad de los poliedros convexos cerrados para representar restricciones, se ve que los mismos no son útiles para representar, por ejemplo, las restricciones de la figura 2.1. Esto se debe a la existencia de una disyunción. Los poliedros convexos son una intersección finita de subespacios generados por relaciones lineales, lo que no da lugar a disyunciones.

Este problema puede ser solucionado de dos maneras. La primera de ellas, es la utilizada en la técnica presentada en [CH78]. En ese trabajo se utiliza la operación *convex hull* (que se describirá más adelante) para obtener un único poliedro convexo cerrado que "abarque" ambas restricciones. En este caso la restricción que se obtendría es: $\{ a = a_0 \wedge ret \geq 0 \wedge ret \leq 1000 \}$, lo que evidentemente representa una pérdida de precisión con respecto a la restricción que se desea representar.

La segunda forma de resolver el problema de representación es extendiéndola a dominios de poliedros convexos cerrados.

**Definición 2.1.2 (Dominio de poliedros convexos cerrados)** *Un dominio de poliedros convexos cerrados de dimensión n es un conjunto de poliedros convexos cerrados de dimensión n.*

```
        int signo(int a)
        {
            int ret;
            if (a >= 0)
            {
                ret = 0;
            }
            else
            {
                 ret = 1000;
            }
1:          return ret;
        }
```

| | | |
|---|---|---|
| 1: | antes | $\{\ a = a_0 \wedge a \geq 0 \wedge ret = 0\ \} \cup \{\ a = a_0 \wedge a < 0 \wedge ret = 1000\ \}$ |
| | después | $\{a = a_0 \wedge a \geq 0 \wedge ret = 0 \wedge ret = retValue\} \cup$ $\{a = a_0 \wedge a < 0 \wedge ret = 1000 \wedge ret = retValue\ \}$ |

Figura 2.1: Unión de poliedros convexos cerrados

Claramente, utilizando dominios de poliedros convexos cerrados, la restricción $\{\ a = a_0 \wedge a \geq 0 \wedge ret = 0\ \} \cup \{\ a = a_0 \wedge a < 0 \wedge ret = 1000\ \}$ es naturalmente representable si se considera que la restricción está representada por la disyunción de cada poliedro del dominio:

$$\{\{a = a_0 \wedge a \geq 0 \wedge ret = 0\}, \{a = a_0 \wedge a < 0 \wedge ret = 1000\}\}$$

La desventaja de esta solución radica en que se aumenta la complejidad de la estructura de representación y, en consecuencia, de las operaciones para manipularla.

### 2.1.1. Operaciones sobre poliedros convexos cerrados

A continuación se definirá una serie de conceptos y operaciones sobre poliedros convexos cerrados. Estos conceptos y operaciones serán posteriormente utilizados en la descripción de la técnica presentada en este trabajo.

Cabe aclarar que los poliedros serán considerados *tipos abstractos de datos*. De esta forma no se presentará ninguna representación particular y sólo se especificará el comportamiento que se espera de las operaciones, sin mostrar algoritmos que las implementen. De hecho, como se puede ver en [CH78], existen al menos dos formas de representar poliedros. Una es la representación implícita utilizada en la definición. La otra, llamada paramétrica, describe el conjunto de puntos (poliedro) como una combinación de "líneas", "vértices" y "rayos". Además, existen algoritmos que permiten la conversión automática de una forma de representación a la otra. La existencia de estas dos representaciones se ve justificada en el hecho de que algunas operaciones sobre poliedros convexos cerrados pueden ser implementadas más eficientemente utilizando una representación, mientras que otras operaciones pueden resultan más eficientes sobre la restante. De hecho, las implementaciones reales de operaciones sobre poliedros como, por ejemplo, *PolyLib*[1] [Loe99] utilizan una doble representación [MRTT53] lo que permite aprovechar lo mejor de cada estructura.

[1] la implementación de la técnica presentada en este trabajo utiliza *PolyLib* para representar y operar sobre dominios de poliedros convexos cerrados

Las definiciones relacionadas a poliedros utilizadas en este trabajo son:

**Poliedro universal:**

El poliedro universal de dimensión $n$ es el conjunto de puntos:

$$P^n_{universal} = \{(x_1, ..., x_n) \in \mathbb{Z}^n\}$$

Existen varias formas de representar dicho poliedro para cada posible dimensión. Por ejemplo, una posible representación para $P^2_{universal}$ se obtiene considerando como matriz y vector a:

$$A = \begin{pmatrix} 0 & 0 \end{pmatrix}$$

$$b = \begin{pmatrix} 1 \end{pmatrix}$$

El poliedro universal es utilizado para representar la restricción { `TRUE` }, presente, por ejemplo, en la figura 1.1. Esta restricción indica que no hay información para el punto del programa correspondiente.

**Poliedro vacío:**

El poliedro vacío, para cualquier dimensión $n$, es simplemente el conjunto de puntos vacío:

$$P^n_{vacio} = \emptyset$$

Existen varias formas de representar dicho poliedro para cada posible dimensión. Por ejemplo, una posible representación para $P^3_{vacio}$ se obtiene considerando como matriz y vector a:

$$A = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$$

$$b = \begin{pmatrix} -1 \end{pmatrix}$$

El poliedro vacío es utilizado para representar la restricción { `FALSE` }, presente, por ejemplo, en la figura 1.2. Esta restricción indica que no existe ninguna combinación válida de valores de variables, o sea el flujo del programa nunca podrá atravesar el punto correspondiente en tiempo de ejecución.

**Inclusión:**

Al considerar a los poliedros como conjuntos de puntos, su inclusión es representada mediante la inclusión de conjuntos estándar. Sean $P_1$ y $P_2$ dos poliedros de dimensión $n$:

$$P_1 \subseteq P_2 \Leftrightarrow \forall x. x \in P_1 \Rightarrow x \in P_2$$

**Intersección:**

Sean $P_1$ y $P_2$ dos poliedros de dimensión $n$, su intersección $P_1 \cap P_2$ es el poliedro que cumple:

$$P_1 \cap P_2 = \{(x_1, ..., x_n) | (x_1, ..., x_n) \in P_1 \wedge (x_1, ..., x_n) \in P_2\}$$

La existencia del poliedro $P_1 \cap P_2$ se demuestra intuitivamente considerando que siempre es posible construir un nuevo poliedro que posea todas las restricciones de $P_1$ y de $P_2$. La matriz $A$ que define este nuevo poliedro (ver definición 2.1.1) tendrá como cantidad de filas a la cantidad de filas de la matriz del poliedro $P_1$ más la cantidad de filas de la matriz del poliedro $P_2$, teniendo en cuenta que cada fila representa una restricción lineal.

***Convex hull:***

Sean $P_1$ y $P_2$ dos poliedros de dimensión $n$, su *convex hull* $(P_1 \uplus P_2)$ es el poliedro que cumple:

$$P_1 \subseteq (P_1 \uplus P_2) \wedge P_2 \subseteq (P_1 \uplus P_2) \wedge (\forall P_3 . P_1 \subseteq P_3 \wedge P_2 \subseteq P_3 \Rightarrow (P_1 \uplus P_2) \subseteq P_3$$

El poliedro resultante de la operación *convex hull* es el menor de todos los poliedros que contienen a $P_1$ y $P_2$.

Esta operación es necesaria porque la unión de poliedros considerada simplemente como la unión de dos conjuntos de puntos, no necesariamente permite obtener un conjunto de puntos representable mediante un poliedro convexo cerrado.

**Extensión de la dimensión en $m$ unidades**

Sea $P_1$ un poliedro de dimensión $n$, la extensión de la dimensión de $P_1$ en $m$ unidades, con $m > 0$ es otro poliedro que cumple que:

$$P_1 \nearrow m = \{(x_1, ..., x_n, x_{n+1}, ..., x_{n+m}) \in \mathbb{Z}^{n+m} | (x_1, ..., x_n) \in P_1\}$$

**Eliminación existencial de una dimensión**

Sea $P_1$ un poliedro de dimensión $n$, la eliminación de la dimensión $i$ con $1 \leq i \leq n$ es otro poliedro de dimensión $n-1$, que cumple que:

$$P_1 \searrow i = \{(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n) \in \mathbb{Z}^{n-1} | \exists x_i . (x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n) \in P_1\}$$

La eliminación existencial de una dimensión posee una interesante propiedad. Esta es que al eliminar una dimensión se conservan todas las relaciones que existían previamente, aun las implícitas, entre las variables de las dimensiones que no han sido eliminadas. Por

ejemplo, supongamos que se cuenta con el poliedro $P_1 = \{\ a = b\ \wedge\ b = c\ \}$. Este poliedro contiene implícita la relación $a = c$. Si se elimina la dimensión correspondiente a la variable $b$ (dimensión número 2), se obtiene como resultado el poliedro $\{\ a = c\ \}$.

Este fenómeno es consecuencia de la propia definición del operador. Por ejemplo, si se considera el poliedro $P_1$, se observa que está compuesto por siguiente el conjunto infinito de elementos: $\{..., (-2,-2,-2), (-1,-1,-1), (0,0,0), (1,1,1), (2,2,2), ...\}$. El poliedro $P_1 \searrow 2$ está compuesto por los pares de elementos $(x_i, z_i)$ tales que exista un $y_i$ que haga que $(x_i, y_i, z_i) \in P_1$. La única posibilidad de que se cumpla $(x_i, y_i, z_i) \in P_1$ es que se pueda establecer la relación $x_i = z_i$, lo que en definitiva convierte a $P_1 \searrow 2$ en el poliedro $\{\ a = c\ \}$.

De la misma forma que ocurre con el ejemplo anterior si se tiene el poliedro $P_2 = \{\ a < b \wedge b < c\ \}$, y se elimina la dimensión correspondiente a la variable $b$ se obtendrá el poliedro $\{\ a + 1 < c\ \}$, lo que corresponde a la relación implícita entre las variables $a$ y $c$ en $P_2$.

Por el contrario a los ejemplos anteriores, existe la posibilidad de que al eliminar una dimensión se pierdan todas las relaciones y se obtenga el poliedro universal. Por ejemplo, si se tiene el poliedro $P_3 = \{\ a \leq b \wedge c \leq b\ \}$, y se elimina la dimensión correspondiente a la variable $b$ se obtendrá el poliedro $\{$ `TRUE` $\}$, lo que indica que no hay relaciones implícitas que se conservan.

Como último ejemplo del fenómeno que produce el operador $\searrow$ se puede observar que si se tiene el poliedro $P_4 = \{\ a \geq b \wedge b = 4\ \}$ y se elimina la dimensión correspondiente a la variable $b$, dimensión número 2, se obtiene el poliedro $\{\ a \geq 4\ \}$. En la figura 2.2 puede verse como existe una interpretación geométrica para el operador $\searrow$. Dicha interpretación es que el resultado de aplicar el operador $\searrow$ es el mismo que el de realizar la proyección de la figura n-dimensional que representa el poliedro original sobre las $n-1$ dimensiones restantes (las dimensiones sobre las que no se realiza la eliminación existeencial). Si bien los poliedros son puntos en el espacio, en la figura se ha optado por una representación continua para facilitar su comprensión.

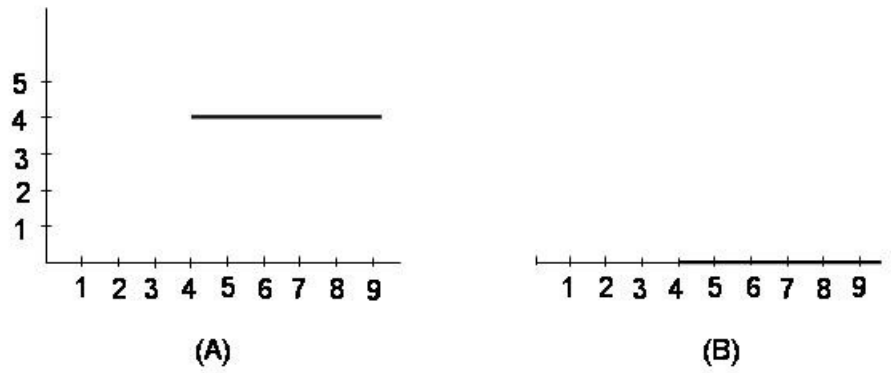


Figura 2.2: (A) Diagrama que representa el poliedro $P_4 = \{\ a \geq b \wedge b = 4\ \}$. (B) Diagrama que representa el poliedro $P_4 \searrow 2$.

**Olvido de una dimensión**

Sea $P_1$ un poliedro de dimensión $n$, el olvido de la dimensión $i$ con $1 \leq i \leq n$ es otro poliedro de dimensión $n$, que cumple que:

$$P_1 \parallel i = \{(x_1, ..., x_{i-1}, x_k, x_{i+1}, ..., x_n) \in \mathbb{Z}^n | \exists x_i.(x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n) \in P_1\}$$

Intuitivamente esta operación corresponde al poliedro que "olvida" las restricciones existentes para determinada dimensión y permite que en la misma pueda existir cualquier valor. Desde el punto de vista de la implementación, este operador es análogo a eliminar la dimensión que se olvida ($\searrow$), luego extender la dimensión en una unidad ($\nearrow 1$) y luego reordenar los índices para para que la nueva dimensión ocupe el lugar de la original.

### 2.1.2. Operaciones sobre dominios de poliedros convexos cerrados

Las definiciones de la sección 2.1.1 pueden ser extendidas a dominios de poliedros convexos cerrados (ver definición 2.1.2). Varias de estas definiciones son repeticiones de las operaciones sobre poliedros adaptadas a dominios.

**Dominio universal:**

El dominio universal de dimensión $n$ es simplemente el conjunto que contiene únicamente el poliedro universal de dimensión $n$.

$$D^n_{universal} = \{P^n_{universal}\}$$

**Dominio vacío:**

El dominio vacío, para cualquier dimensión $n$, es:

$$D^n_{vacio} = \{P^n_{vacio}\}$$

**Inclusión:**

La inclusión entre dominios está definida en [BHZ04] de la siguiente manera:

$$D_1 \subseteq D_2 \Leftrightarrow \forall P_1 \in D_1 : \exists P_2 \in D_2.P_1 \subseteq P_2$$

**Intersección:**

Sean $D_1$ y $D_2$ dos dominios de dimensión $n$, su intersección $D_1 \cap D_2$ es el dominio que cumple:

$$D_1 \cap D_2 = \{(x_1, ..., x_n) | (x_1, ..., x_n) \in D_1 \wedge (x_1, ..., x_n) \in D_2\}$$

En la fórmula anterior se ha realizado un abuso de notación en la utilización de la relación $\in$ para referirse a la pertenencia de un punto a un dominio. En realidad un punto $x$ pertenece a un dominio si pertenece a alguno de los poliedros que contiene el dominio:

$$(x_1, ..., x_n) \in D_1 \Leftrightarrow \exists P_i : P_i \in D_1 \wedge (x_1, ..., x_n) \in P_i$$

La existencia del dominio $D_1 \cap D_2$ se demuestra intuitivamente realizando de forma combinatoria la intersección de cada uno de los poliedros que componen el dominio $D_1$ con los que componen el dominio $D_2$, y uniendo los poliedros formando un único conjunto. Por ejemplo, si $D_1 = \{P_1, P_2, P_3\}$ y $D_2 = \{P_4, P_5\}$, entonces: $D_1 \cap D_2 = \{(P_1 \cap P_4), (P_2 \cap P_4), (P_3 \cap P_4), (P_1 \cap P_5), (P_2 \cap P_5), (P_3 \cap P_5)\}$.

**Unión:**

A diferencia de lo que ocurre con los poliedros convexos cerrados, la unión de dominios sí es una operación válida.

Sean $D_1$ y $D_2$ dos dominios de dimensión $n$, su unión $D_1 \cup D_2$ es el dominio que cumple:

$$D_1 \cup D_2 = \{(x_1, ..., x_n) | (x_1, ..., x_n) \in D_1 \vee (x_1, ..., x_n) \in D_2\}$$

La existencia del dominio $D_1 \cup D_2$ se demuestra trivialmente a partir de la definición 2.1.2

***Convex hull:***

La operación *convex hull* en dominios es unaria, y arroja como resultado un único poliedro convexo cerrado.

Sea $D^n$ un dominio de dimensión $n$ formado por los poliedros $P_1^n, P_2^n, ..., P_k^n$, su *convex hull* $\uplus D^n$ es el poliedro convexo cerrado que cumple:

$$P_1^n \subseteq \uplus D^n \wedge P_2 \subseteq \uplus D^n \wedge ... \wedge P_k^n \subseteq \uplus D^n \wedge (\forall P_u^n . P_1^n \subseteq P_u^n \wedge P_2^n \subseteq P_u^n \wedge ... \wedge P_k^n \subseteq P_u^n \Rightarrow \uplus D^n \subseteq P_u^n$$

El poliedro resultante de la operación *convex hull* es el menor de todos los poliedros que contienen a los poliedros $P_1^n, P_2^n, ..., P_k^n$ que conforman a $D^n$.

**Extensión de la dimensión en $m$ unidades**

Sea $D_1$ un dominio de dimensión $n$, la extensión de la dimensión de $D_1$ en $m$ unidades, con $m > 0$ es otro dominio que cumple que:

$$D_1 \nearrow m = \{(x_1, ..., x_n, x_{n+1}, ..., x_{n+m}) \in \mathbb{Z}^{n+m} | (x_1, ..., x_n) \in D_1\}$$

#### Eliminación existencial de una dimensión

Sea $D_1$ un dominio de dimensión $n$, la eliminación de la dimensión $i$ con $1 \leq i \leq n$ es otro dominio de dimensión $n-1$, que cumple que:

$$D_1 \searrow i = \{(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n) \in \mathbb{Z}^{n-1} | \exists x_i . (x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n) \in D_1\}$$

#### Olvido de una dimensión

Sea $D_1$ un dominio de dimensión $n$, el olvido de la dimensión $i$ con $1 \leq i \leq n$ es otro dominio de dimensión $n$, que cumple que:

$$D_1 \parallel i = \{(x_1, ..., x_{i-1}, x_k, x_{i+1}, ..., x_n) \in \mathbb{Z}^{n} | \exists x_i . (x_1, ..., x_{i-1}, x_i, x_{i+1}, ..., x_n) \in D_1\}$$

## 2.2. Reticulado

Los conjuntos parcialmente ordenados y los reticulados completos son dos conceptos fundamentales en el análisis de programas. Es por esto que, si bien pueden ser encontrados en la mayoría de la literatura relacionada como por ejemplo [NNH99], se hará una breve descripción a continuación.

#### Conjunto parcialmente ordenado

Un orden parcial es una relación $\sqsubseteq \subseteq L \times L$ que es reflexiva ($\forall l : l \sqsubseteq l$), transitiva ($\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$) y anti-simétrica ($\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$).

Un conjunto parcialmente ordenado $(L, \sqsubseteq)$ es un conjunto $L$ sobre el cual se define un orden parcial $\sqsubseteq$.

Un subconjunto $Y$ de $L$ tiene una cota superior $l \in L$ si $\forall l' \in Y : l' \sqsubseteq l$. Una menor cota superior $l$ de $Y$ es una cota superior de $Y$ que cumple que $l \sqsubseteq l_0$ siempre que $l_0$ sea otra cota superior de $Y$. Análogamente, una mayor cota inferior $l$ de $Y$ es una cota inferior de $Y$ que cumple que $l_0 \sqsubseteq l$ siempre que $l_0$ sea otra cota inferior de $Y$. Los subconjuntos $Y$ de un conjunto parcialmente ordenado $L$ no necesariamente siempre tienen una menor cota superior, o una mayor cota inferior, pero cuando sí las tienen son siempre únicas (debido a que $\sqsubseteq$ es anti-simétrica) y se denotan $\bigsqcup Y$ y $\bigsqcap Y$ respectivamente.

#### Reticulado completo

Un reticulado completo es un conjunto parcialmente ordenado $(L, \sqsubseteq)$ que cumple la propiedad de que todo posible subconjunto $Y$ tiene menor cota superior y mayor cota inferior.

Un reticulado completo se denota $(L, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$, donde $L$ es el conjunto parcialmente ordenado de elementos, $\sqsubseteq$ es el orden parcial, $\bigsqcup$ es el operador que define la menor cota superior de un subconjunto de $L$, $\bigsqcap$ es el operador que define la mayor cota inferior

de un subconjunto de $L$, $\bot = \bigsqcup \emptyset = \bigsqcap L$ es el menor elemento de $L$ y $\top = \bigsqcap \emptyset = \bigsqcup L$ es el mayor elemento de $L$.

A los efectos de este trabajo el concepto principal es que un conjunto de dominios de poliedros convexos cerrados es un conjunto parcialmente ordenado, considerando la relación $\subseteq$, y particularmente también es un reticulado completo, considerando:

$$\bigsqcup Y = \bigcup_{y \in Y} y$$

y

$$\bigsqcap Y = \bigcap_{y \in Y} y$$

o sea que la menor cota superior de un conjunto de dominios se obtiene realizando la disyunción de cada uno de ellos, y la mayor cota inferior se obtiene realizando la intersección de cada uno de ellos. Además el menor elemento del conjunto de todos los dominios es $D_{vacio}$ y el mayor elemento es $D_{universal}$.

# Capítulo 3

# El análisis en programas de números enteros

En este capítulo se presentan los principales aportes de la tesis. Estos aportes consisten básicamente en el desarrollo de tres técnicas. La primera de ellas es la adaptación de la técnica presentada en [CH78] para trabajar con dominios de poliedros convexos cerrados. En este sentido, en la sección 3.1 se presenta el lenguaje de programas mono-procedurales que realizan operaciones sobre números enteros, y el la sección 3.2 se presenta el análisis correspondiente.

La segunda y tercera técnica se presentan en la sección 3.3 y consisten en extender la técnica de la sección 3.2 para considerar programas que realizan invocaciones a procedimientos. En particular, la segunda técnica aborda la invocación a procedimientos mediante la simulación de *inlinig* y la tercera mediante análisis simbólico. Cabe destacar que estas dos últimas técnicas también utilizan dominios de poliedros convexos cerrados para representar las restricciones.

Debido a que las tres técnicas presentadas se basan en la construcción de ecuaciones de *Data Flow*, en la sección 3.4 se presenta un método para resolver dichos sistemas de ecuaciones.

Finalmente, en la sección 3.5 se presentan algunas ideas para obtener información de relaciones no lineales de forma tal de extender las capacidades de las tres técnicas presentadas.

## 3.1. El lenguaje de los programas de números enteros

Los programas sobre los que inicialmente se realizará el análisis son los escritos utilizando la sintaxis de la figura 3.1

La figura 3.2 muestra las abreviaturas para las categorías sintácticas que podrán ser encontradas en el resto del documento.

La categoría sintáctica ExpresiónNoLineal no es expandida debido a que, a los efectos del análisis, todas sus posibles expresiones tienen asociada la misma semántica. Como se verá más adelante, al asignar cualquier expresión no lineal a una variable se pierde toda la información que se pueda haber obtenido sobre ella. Ejemplos de expresiones no lineales

| | | |
|---|---|---|
| Programa | → | SentenciaEtiquetada |
| SentenciaEtiquetada | → | (Etiqueta, Sentencia) |
| SentenciaEtiquetada | → | (Etiqueta, Sentencia); SentenciaEtiquetada |
| Etiqueta | → | $n$ con $n \in \mathbb{N}$ |
| Sentencia | → | `if (` ExpresiónBooleana `) goto` Etiqueta |
| Sentencia | → | `goto` Etiqueta |
| Sentencia | → | `return` |
| Sentencia | → | `return` Operando |
| Sentencia | → | Asignación |
| ExpresiónBooleana | → | *variableLocal* OperadorBooleano Operando |
| ExpresiónBooleana | → | Operando OperadorBooleano *variableLocal* |
| OperadorBooleano | → | `==` |
| OperadorBooleano | → | `!=` |
| OperadorBooleano | → | `>=` |
| OperadorBooleano | → | `>` |
| OperadorBooleano | → | `<=` |
| OperadorBooleano | → | `<` |
| Asignación | → | *variableLocal* `=` ExpresiónZ<br>donde *variableLocal* es una cadena de caracteres |
| ExpresiónZ | → | $z$ con $z \in \mathbf{Z}$ |
| ExpresiónZ | → | *variableLocal* |
| ExpresiónZ | → | `-` *variableLocal* |
| ExpresiónZ | → | *variableLocal* OperadorZ Operando |
| ExpresiónZ | → | Operando OperadorZ *variableLocal* |
| ExpresiónZ | → | *variableLocal* `*` $z$ |
| ExpresiónZ | → | $z$ `*` *variableLocal* |
| ExpresiónZ | → | ExpresiónNoLineal |
| OperadorZ | → | `+` |
| OperadorZ | → | `-` |
| Operando | → | *variableLocal* |
| Operando | → | $z$ |

Figura 3.1: Sintaxis del lenguaje.

| | | |
|---|---|---|
| Programa | = | Prgm |
| SentenciaEtiquetada | = | StncEtq |
| Etiqueta | = | Etq |
| Sentencia | = | Stnc |
| ExpresiónBooleana | = | ExprB |
| OperadorBooleano | = | OpB |
| Asignación | = | Asig |
| ExpresiónZ | = | ExprZ |
| OperadorZ | = | OpZ |
| Operando | = | Opnd |
| ExpresiónNoLineal | = | ExprNL |

Figura 3.2: Abreviaturas de categorías sintácticas.

pueden ser aquellas que involucren la operación potencia, raíz, módulo, el producto en entre variables, etc.

Es necesario hacer algunas aclaraciones con respecto a este lenguaje. Si se observa, existen dos formas de alterar el flujo del programa, una es mediante el salto condicional "`if (` ExpresiónBooleana `)` `goto` Etiqueta" y la otra mediante el salto incondicional "`goto` Etiqueta". Esta forma de alterar el flujo de un programa dista mucho de las que es posible encontrar en programas escritos con lenguajes de programación modernos, donde es común encontrar estructuras de control como `while`, `for`, etc. Esto se debe a que el análisis presentado está diseñado para trabajar sobre una *representación intermedia*. Es decir, el lenguaje sobre el que se realiza el análisis no está ligado a ninguna sintaxis concreta, sino que por el contrario está fuertemente inspirado en *Jimple* [VRHS+99], que es una representación intermedia del lenguaje Java. De esta forma se logra abstraer los detalles de la sintaxis concreta y facilitar el análisis.

Una última aclaración respecto al lenguaje es que la técnica no contempla ninguna verificación del estilo semántica, como pueden ser verificación de tipos, o verificación de inicialización previa de las variables leídas. Esto se debe a que el análisis utiliza una representación intermedia que se asume obtenida a partir de programas que superen exitosamente el chequeo sintáctico sobre la sintaxis concreta y posteriormente los chequeos semánticos correspondientes. En este sentido se asumirá que los programas a analizar siempre están bien formados y cumplen con todas las condiciones de correctitud semántica necesarias. Entre las condiciones que se asumen es posible destacar que los números naturales que representan las etiquetas de cada sentencia no deben repetirse, de lo contrario se presentarían ambigüedades en el momento de considerar alteraciones de flujo; asimismo, se asumirá que las etiquetas que representan destinos de los saltos son efectivamente etiquetas existentes; también se asumirá que los programas inicializan mediante definiciones todas las variables que luego son referenciadas; etc.

## 3.2. El análisis para programas de números enteros

El análisis será presentado como un análisis *Data Flow*. Existe vasta literatura que describe las características de este tipo de análisis sobre programas, entre la que podremos destacar a [NNH99].

En los análisis *Data Flow* resulta muy conveniente abstraer los programas a analizar utilizando grafos. De esta forma se modelan fácilmente las alternativas del flujo del programa a través de las instrucciones.

### 3.2.1. Grafo de programas de números enteros

El grafo de un programa se obtiene a partir de las siguientes definiciones:

**Definición 3.2.1 (Conjunto de nodos del grafo de un programa $P$)** *El conjunto de nodos del grafo de un programa $P$ se denota $G^P_{nodos}$ y se define de la siguiente manera:*

$$\begin{array}{lcl} G^P_{nodos} & = & \{Etq\} \text{ si } P = (Etq, Stnc) \\ G^P_{nodos} & = & \{Etq\} \cup G^{P'}_{nodos} \text{ si } P = (Etq, Stnc); P' \end{array}$$

**Definición 3.2.2 (Conjunto de ejes del grafo de un programa $P$)** *Un eje del grafo de un programa es un par compuesto por dos etiquetas $(Etq_a, Etq_b)$. El conjunto de ejes del grafo de un programa $P$ se denota $G^P_{ejes}$ y se define de la siguiente manera:*

$$G^P_{ejes} = \emptyset \text{ si } P = (Etq, Stnc)$$

$$\begin{array}{lcl} G^P_{ejes} & = & \{(Etq_1, Etq_2), (Etq_1, Etq_3)\} \cup G^{(Etq_2, Stnc)P'}_{ejes} \\ & & \text{si } P = (Etq_1, \texttt{if (}ExprB\texttt{) goto } Etq_3); (Etq_2, Stnc)P' \end{array}$$

$$G^P_{ejes} = \{(Etq_1, Etq_2)\} \cup G^{(Etq_2, Stnc)P'}_{ejes} \text{ si } P = (Etq_1, Asig); (Etq_2, Stnc)P'$$

$$G^P_{ejes} = \{(Etq_1, Etq_2)\} \cup G^{P'}_{ejes} \text{ si } P = (Etq_1, \texttt{goto } Etq_2)P'$$

$$G^P_{ejes} = G^{P'}_{ejes} \text{ si } P = (Etq_1, \texttt{return})P'$$

$$G^P_{ejes} = G^{P'}_{ejes} \text{ si } P = (Etq_1, \texttt{return } op)P'$$

**Definición 3.2.3 (Grafo de un programa $P$)** *El grafo de un programa $P$, denotado $G^P$ es el grafo dirigido que contiene como conjunto de nodos a $G^P_{nodos}$, y como conjunto de ejes a $G^P_{ejes}$.*

Si bien el grafo sólo brinda información sobre las etiquetas del programa, estas permiten identificar unívocamente sentencias. Por lo cual resulta equivalente considerar que los nodos del grafo son etiquetas o sentencias. En la figura 3.4 puede observarse un ejemplo de grafo de programa.

### 3.2.2. Ecuaciones del análisis *Data Flow*

Las restricciones lineales que se pretende descubrir, antes y después de cada sentencia del programa, pueden considerarse las incógnitas del análisis. Luego, es posible construir un sistema de ecuaciones (ecuaciones de *Data Flow*) que tenga la propiedad de que su solución sea la solución del análisis.

```
(1, a = 3);
(2, if a <= 10 goto 5);
(3, ret = 1);
(4, goto 6);
(5, ret = 0);
(6, return ret)
```

(a)

```
int a = 3;
int ret;
if (a > 10)
    ret = 1;
else
    ret = 0;
return ret;
```

(b)

Figura 3.3: (a) Programa sencillo en la sintaxis del análisis (b) versión del programa en utilizando una hipotética sintaxis concreta

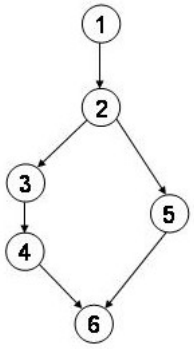


Figura 3.4: Grafo correspondiente al programa (a) de la figura 3.3

La cantidad de incógnitas del análisis dependerá de la cantidad de sentencias que componen el programa a analizar. Además las ecuaciones quedan determinadas por los diferentes tipos de sentencia (definición, salto condicional, etc.) que componen el programa. Es por esto que cada programa a analizar tendrá una cantidad y variedad de ecuaciones diferente, por lo que no es posible presentar el sistema de ecuaciones para encontrar restricciones lineales para todos los programas escritos en el lenguaje dado. Lo que sí es posible definir, es un sistema de reglas que permiten, considerando un programa particular, construir el sistema de ecuaciones cuya solución es el conjunto de restricciones lineales que se busca.

Antes de presentar el sistema de reglas es necesario realizar las siguientes definiciones

**Definición 3.2.4 (Longitud de un programa $P$)** *La longitud de un programa $P$ es un número entero que describe la cantidad de sentencias etiquetadas que lo componen.*

$$\begin{aligned} Long(P) &= 1 \text{ si } P = (Etq, Stnc) \\ Long(P) &= 1 + Long(P') \text{ si } P = (Etq, Stnc); P' \end{aligned}$$

**Definición 3.2.5 (Etiqueta inicial de un programa $P$)** *La etiqueta inicial de un programa $P$ es la etiqueta de la primera sentencia etiquetada que es considerada cuando el programa es ejecutado.*

$$\begin{aligned} Prim(P) &= Etq_1 \text{ si } P = (Etq_1, Stnc) \\ Prim(P) &= Etq_1 \text{ si } P = (Etq_1, Stnc); P' \end{aligned}$$

**Definición 3.2.6 (Conjunto de variables locales $P$)** *El conjunto de variables locales de un programa $P$ se define de la siguiente manera:*

$$\begin{aligned} Vars(P) &= \{variableLocal\} \text{ si } P = (Etq, variableLocal\texttt{=}ExpresinZ) \\ Vars(P) &= \emptyset \text{ si } P = (Etq, Stnc) \wedge Stnc \neq Asig \\ Vars(P) &= Vars(P') \text{ si } P = (Etq, Stnc); P' \wedge Stnc \neq Asig \\ Vars(P) &= \{variableLocal\} \cup Vars(P') \\ &\qquad \text{si } P = (Etq, variableLocal\texttt{=}ExpresinZ); P' \end{aligned}$$

*Cabe mencionar que esta definición se sostiene fuertemente de la condición de correctitud, que se considera asumida, de que todas las variables locales que son leías en alguna sentencia del programa fueron definidas en alguna sentencia previa.*

**Definición 3.2.7 (Conjunto de variables del análisis del programa $P$)** *El conjunto de variables del análisis del programa $P$ es:*

$$VarsAn(P) = Vars(P) \cup \{retValue\}$$

*Se asume que la variable retValue tiene un nombre distinguido que no se utiliza en ningún programa. Se utiliza para representar el valor de retorno del programa.*

El sistema de ecuaciones *Data Flow* para un programa $P$ tendrá $2 * Long(P)$ incógnitas y $2 * Long(P)$ ecuaciones.

Las incógnitas se denotan $Etq_i^{antes}$ y $Etq_i^{después}$ y son los dominios de poliedros convexos cerrados (ver definición 2.1.2) que representan la restricción lineal entre las variables del programa existente antes de la ejecución de la sentencia etiquetada $Etq_i$ y después de la ejecución de dicha sentencia, respectivamente.

Para representar las restricciones lineales los dominios de poliedros convexos cerrados utilizados serán ligeramente diferentes a los definidos anteriormente. La diferencia radica en que cada dimensión del poliedro se considera "mapeada" a una variable del análisis, o sea un elemento de $VarsAn(P)$. De esta forma, en lugar de referirse a una dimensión particular utilizando un índice, se lo hará utilizando el nombre de una variable. Además es necesario asumir que el "mapeo" se mantiene uniformemente a lo largo del análisis, para que las operaciones que involucran más de un dominio sean consistentes.

Las reglas que permiten derivar el sistema de ecuaciones se denotan $R_i^p$ (donde $i$ depende del tipo de sentencia asociada a la regla) y se agrupan en dos conjuntos. El primer conjunto de reglas es el que permite obtener ecuaciones que relacionan las incógnitas que representan restricciones después de ejecutar una sentencia, con la restricción existente antes de ejecutarla:

$R_1^P$. $Etq_i^{después} = Etq_i^{antes}$
si $Etq_i$ corresponde a una sentencia del tipo `if (`$ExprB$`) goto` $Etq$

$R_2^P$. $Etq_i^{después} = Etq_i^{antes}$

si $Etq_i$ corresponde a una sentencia del tipo `goto` $Etq$

$R_3^P$. $Etq_i^{después} = Etq_i^{antes}$
si $Etq_i$ corresponde a una sentencia del tipo `return`

$R_4^P$. $Etq_i^{después} = Etq_i^{antes} \cap nuevoDominio(retValue = op, VarsAn(P))$
si $Etq_i$ corresponde a una sentencia del tipo `return op`

$R_5^P$. $Etq_i^{después} = (((Etq_i^{antes} \nearrow tmp \cap D_{def}) \parallel variableLocal) \cap D_{bind}) \searrow tmp$

donde:
$tmp$ es una variable 'fresca', o sea $tmp \notin VarsAn(P)$
$D_{def} = nuevoDominio(tmp = ExprZ, VarsAn(P) \cup \{tmp\})$
$D_{bind} = nuevoDominio(variableLocal = tmp, VarsAn(P) \cup \{tmp\})$

si $Etq_i$ corresponde a una sentencia del tipo $variableLocal = ExprZ \wedge$
$ExprZ \neq ExprNL$

$R_6^P$. $Etq_i^{después} = Etq_i^{antes} \parallel variableLocal$
si $Etq_i$ corresponde a una sentencia del tipo $variableLocal = ExprNL$

El segundo conjunto contiene las reglas que relacionan las incógnitas que representan restricciones antes de ejecutar una sentencia, con las restricciones después de ejecutar cada una de las sentencias que la anteceden inmediatamente en el flujo del programa.

$R_7^P$. $Etq_i^{antes} = D_{universal}^{VarsAn(p)}$ si $Etq_i = Prim(P)$

$R_8^P$. $Etq_i^{antes} = \bigcup_{(j,i) \in G_{ejes}^P} Etq_j^{después}$ si $Etq_i \neq Prim(P)$

Las operaciones y conceptos utilizados en las reglas fueron todos definidos en la sección 2.1.1 a excepción de la expresión $nuevoDominio(expr, vars)$. Se considerará que dicha expresión corresponde a un nuevo dominio que se construye y tiene como dimensión la cantidad de elementos de $vars$, además representa la restricción lineal descrita por la expresión $expr$, y es consistente con el mapeo entre variables del programa y dimensiones del dominio subyacente en el análisis. El detalle de cómo la expresión lineal puede ser representada mediante el nuevo dominio construido es omitido, pero resulta intuitivo a partir de las ideas presentadas en la sección 2.1 especialmente a partir de la definición 2.1.1.

La mayoría de las reglas presentadas resultan fácilmente comprensibles. En el primer grupo de reglas, las que relacionan la restricción existente después de ejecutar una sentencia con la restricción existente antes de ejecutarla, puede verse que en presencia de saltos condicionales e incondicionales no se producen modificaciones. Por otro lado, la sentencia `return` da por terminado un programa sin que se devuelva ningún valor, por lo tanto tampoco se modifica la restricción existente antes de dicha sentencia. La regla $R_4^P$ tiene el efecto de 'ligar' el valor de retorno del programa con la variable distinguida $retValue$; de esta forma la restricción resultante puede predicar de forma directa sobre el valor retornado por el programa.

La regla $R_6^P$ tiene el efecto de perder toda información que se podía tener sobre la variable definida. Esta regla constituye una de las fuentes de pérdida de precisión en el análisis. Sabiendo que los dominios de poliedros convexos cerrados no pueden representar relaciones no lineales, lo mejor que se puede hacer para que el análisis se mantenga correcto es actuar conservativamente y simplemente no brindar ningún tipo de información sobre la variable afectada. Luego en la sección 3.5 se verá como es posible representar mediante relaciones lineales algunas expresiones naturalmente no lineales como por ejemplo $a = b \ mod \ c$.

La regla $R_5^P$ es, quizás, la regla más complicada y amerita una explicación. Gran parte de la complejidad de la regla radica en que debe ser capaz de reflejar definiciones donde la variable que se define también es leída. Esta situación es muy común en sentencias que incrementan contadores, como por ejemplo, `i = i + 1`. La regla utiliza la variable 'fresca' *tmp*. Se asume que *tmp* será un nombre distinguido y no podrá ser utilizado como nombre para las variables del programa a analizar. Esta variable es utilizada para 'almacenar' temporalmente el valor de la expresión. En $D_{def}$ queda establecida una relación de igualdad entre la expresión que se asigna y la nueva variable. Además en necesario extender la dimensión de $Etq_i^{antes}$ debido a que los dominios deben ser compatibles (tener la misma dimensión y el mismo mapeo de dimensiones a variables) para poder operar con ellos. Luego de establecida la igualdad entre la variable temporal y la expresión a asignar, es necesario 'olvidar' toda la información que se tenía sobre la variable que realmente se está definiendo para luego sí, utilizando $D_{bind}$, establecer la relación que define la sentencia. Finalmente, se elimina la variable temporal.

Otra forma de presentar la misma información que proveen las reglas del primer conjunto $(R_1^P, R_2^P, R_3^P, R_4^P, R_5^P, R_6^P)$ es mediante la definición de una función llamada *función de transferencia*. La función de transferencia recibe como argumento un dominio y una sentencia, y devuelve otro dominio. El dominio que cumple el rol de argumento corresponde a la restricción antes de ejecutar la sentencia que también se recibe como argumento $(Etq_i^{antes})$, y el dominio que representa el valor de retorno corresponde a la restricción existente después de ejecutar la sentencia recibida como argumento $(Etq_i^{después})$. A partir de las reglas $R_1^P, R_2^P, R_3^P, R_4^P, R_5^P, R_6^P$ se puede inferir la siguiente función de transferencia:

$$f(d,s) = \begin{cases} d & \text{si } s = \texttt{if (}ExprB\texttt{) goto } Etq \\ d & \text{si } s = \texttt{goto } Etq \\ d & \text{si } s = \texttt{return} \\ d \cap nd(retValue = op, VarsAn(P)) & \text{si } s = \texttt{return } op \\ (((d \nearrow t \cap D_{def}) \parallel v) \cap D_{bind}) \searrow t & \text{si } s = v \texttt{ = } ExprZ \wedge ExprZ \neq ExprNL \\ d \parallel v & \text{si } s = v \texttt{ = } ExprNL \end{cases}$$

Donde $d$ cumple el rol de $(Etq_i^{antes})$, s es la sentencia correspondiente a la etiqueta $Etq_i$, la expresión $nd(retValue = op, VarsAn(P))$ es una abreviatura para $nuevoDominiod(retValue = op, VarsAn(P))$, $D_{def}$ y $D_{bind}$ son los dominios que se indica en la regla $R_5^P$ y $t$ es una variable 'fresca' análoga a la variable *tmp* de la regla $R_5^P$.

En el segundo grupo de reglas, las que relacionan las restricciones antes de ejecutar una sentencia con las restricciones después de ejecutar cada una de las sentencias que la anteceden inmediatamente en el flujo del programa, puede verse que para la primera sentencia del programa se fija como restricción el dominio universal. Esto indica que no

existe información antes de ejecutar la mencionada sentencia. Cabe destacar que aunque existan saltos desde alguna sentencia del programa a la sentencia $Prim(P)$ esto no puede aportar ningún tipo de información ya que se debería realizar la disyunción con el dominio $D_{universal}$. Esto es análogo a realizar la disyunción de una tautología con cualquier otra fórmula en lógica proposicional.

Para las sentencias que no sean la primera es necesario realizar una disyunción de cada una de las posibilidades que define el flujo del programa. De esta manera se actúa conservativamente y se consideran todas las variantes que pueden presentarse al ejecutar el programa.

La regla $R_8^P$ resulta clave para catalogar el análisis *Data Flow* en *forward* y en *may*. Es decir, a partir de la regla $R_8^P$ se puede decir que el análisis que aquí se presenta es un análisis *Data Flow* del tipo *forward* y *may*. Es de tipo *forward* porque, al analizar un punto del programa antes de una sentencia, se consideran los valores obtenidos para las sentencias que la anteceden inmediatamente en el flujo del programa, o sea el análisis va hacia adelante, de la misma forma que lo hace la ejecución del programa. Por otro lado, se está en presencia de un análisis *may* porque la información proveniente de cada sentencia inmediatamente antecesora es unida en lugar de intersecada. Esto refleja que el análisis es conservativo pudiendo en algunos casos considerar situaciones que en tiempo de ejecución no son alcanzables.

A modo de ejemplo se presenta el programa de la figura 3.3. En la figura 3.4 se presenta el grafo correspondiente. Luego, la figura 3.5 presenta el conjunto de ecuaciones que se obtiene cuando se aplica el sistema de reglas a dicho programa.

La figura 3.6 muestra la solución al conjunto de ecuaciones, o sea, el resultado del análisis de restricciones lineales. Un estudio de cómo obtener soluciones para el sistema de ecuaciones es presentado en la sección 3.4

### 3.2.3. Incrementando la precisión del análisis

Las restricciones presentadas en la figura 3.6 son correctas, en el sentido de que podría demostrarse que si se considera como valuaciones para las variables `a` y `ret` a los valores que dichas variables pueden adoptar en tiempo de ejecución, las restricciones evaluarían siempre a verdadero. Además, como se demostró en la proposición 1.1.1, la obtención de restricciones de precisión absoluta no es computable. Teniendo esto en cuenta, podría resultar comprensible conformarse con el resultado obtenido.

Sin embargo, existe la posibilidad de realizar una pequeña modificación a la regla $R_8^P$ para obtener resultados mucho más precisos. La regla se puede definir de la siguiente manera:

$$R_8^P. \quad Etq_i^{antes} = \bigcup_{(j,i) \in G_{ejes}^P} procCond(i, Etq_j) \text{ si } Etq_i \neq Prim(P)$$

donde:

$$
\begin{aligned}
Etq_1^{antes} &= D_{universal}^{\{a,\ ret,\ retValue\}} \\
Etq_1^{después} &= (((Etq_1^{antes} \nearrow tmp \cap D_1) \parallel a) \cap D_2) \searrow tmp \\
&\quad D_1 = nuevoDominio(tmp = 3, \{a,\ ret,\ retValue,\ tmp\}) \\
&\quad D_2 = nuevoDominio(a = tmp, \{a,\ ret,\ retValue,\ tmp\}) \\
Etq_2^{antes} &= Etq_1^{después} \\
Etq_2^{después} &= Etq_2^{antes} \\
Etq_3^{antes} &= Etq_2^{después} \\
Etq_3^{después} &= (((Etq_3^{antes} \nearrow tmp \cap D_3) \parallel ret) \cap D_4) \searrow tmp \\
&\quad D_3 = nuevoDominio(tmp = 1, \{a,\ ret,\ retValue,\ tmp\}) \\
&\quad D_4 = nuevoDominio(ret = tmp, \{a,\ ret,\ retValue,\ tmp\}) \\
Etq_4^{antes} &= Etq_3^{después} \\
Etq_4^{después} &= Etq_4^{antes} \\
Etq_5^{antes} &= Etq_2^{después} \\
Etq_5^{después} &= (((Etq_5^{antes} \nearrow tmp \cap D_5) \parallel ret) \cap D_6) \searrow tmp \\
&\quad D_5 = nuevoDominio(tmp = 0, \{a,\ ret,\ retValue,\ tmp\}) \\
&\quad D_6 = nuevoDominio(ret = tmp, \{a,\ ret,\ retValue,\ tmp\}) \\
Etq_6^{antes} &= Etq_2^{después} \cup Etq_5^{después} \\
Etq_6^{después} &= Etq_6^{antes} \cap nuevoDominio(retValue = ret, \{a,\ ret,\ retValue\})
\end{aligned}
$$

Figura 3.5: Ecuaciones de *Data Flow* obtenidas analizando el programa de la figura 3.3

$$
\begin{aligned}
procCond(i, Etq_j) &= Etq_j^{después} \\
&\quad \text{si } Etq_j \text{ no corresponde a una sentencia del tipo} \\
&\quad \texttt{if (}ExprB_h\texttt{) goto } Etq_h \\
procCond(i, Etq_j) &= Etq_j^{después} \cap nuevoDominio(\neg ExprB_h, VarsAn(p)) \\
&\quad \text{si } Etq_j \text{ corresponde a una sentencia del tipo} \\
&\quad \texttt{if (}ExprB_h\texttt{) goto } Etq_h \wedge Etq_h \neq Etq_i \\
procCond(i, Etq_j) &= Etq_j^{después} \cap nuevoDominio(ExprB_h, VarsAn(p)) \\
&\quad \text{si } Etq_j \text{ corresponde a una sentencia del tipo} \\
&\quad \texttt{if (}ExprB_h\texttt{) goto } Etq_h \wedge Etq_h = Etq_i
\end{aligned}
$$

| | | |
|---|---|---|
| 1: | antes | $\{$ `TRUE` $\}$ |
| | después | $\{\ a = 3\ \}$ |
| 2: | antes | $\{\ a = 3\ \}$ |
| | después | $\{\ a = 3\ \}$ |
| 3: | antes | $\{\ a = 3\ \}$ |
| | después | $\{\ a = 3 \wedge ret = 1\ \}$ |
| 4: | antes | $\{\ a = 3 \wedge ret = 1\ \}$ |
| | después | $\{\ a = 3 \wedge ret = 1\ \}$ |
| 5: | antes | $\{\ a = 3\ \}$ |
| | después | $\{\ a = 3 \wedge ret = 0\ \}$ |
| 6: | antes | $\{\ a = 3 \wedge ret = 1\ \} \cup \{\ a = 3 \wedge ret = 0\ \}$ |
| | después | $\{\ a = 3 \wedge ret = 1 \wedge retValue = ret\ \} \cup$ $\{\ a = 3 \wedge ret = 0 \wedge retValue = ret\ \}$ |

Figura 3.6: Solución al sistema de ecuaciones

Esta nueva regla, no sólo relaciona las restricciones antes de ejecutar una sentencia con las restricciones después de ejecutar cada una de las inmediatas antecesoras en el flujo del programa, sino que además tiene en cuenta las condiciones que pueden hacer que el flujo del programa cambie en presencia de saltos condicionales. Si la sentencia predecesora no es un salto condicional, no se brinda más información que la obtenida después de la ejecución de dicha sentencia. Si la sentencia predecesora sí es un salto condicional, se distinguen dos casos. El primero de ellos se produce cuando estamos en presencia de la instrucción siguiente al salto condicional, lo que indica que el salto condicional no fue tomado. En este caso, se cuenta con más información que la que correspondería en el caso anterior, entonces puede incorporarse a la restricción la negación de la expresión booleana. El segundo caso corresponden a la situación en la que la sentencia a considerar corresponde al destino del salto condicional. En este caso simplemente se incorpora a la restricción la expresión booleana de la condición.

En esta nueva regla nuevamente se utiliza la expresión:

$nuevoDominio(expr, vars)$

Aquí también resulta intuitiva la construcción del dominio a partir de la expresión booleana considerando la definición 2.1.1. Sin embargo, un punto merece ser aclarado y es el que se tiene al representar mediante dominios a las relaciones de desigualdad del tipo $\neq$ (definidas por el operador booleano "`!=`"). En este caso la relación del tipo $\alpha \neq \beta$ puede interpretarse como la relación equivalente $\alpha > \beta \vee \alpha < \beta$, y viendo la relación de esta forma su representación mediante dominio de poliedros convexos cerrados es directa.

Finalmente, en la figura 3.7 se puede ver que el resultado del análisis del programa de la figura 3.3 que se obtiene considerando el nuevo sistema de reglas es mucho más preciso que el obtenido mediante el sistema de reglas anterior.

| | | |
|---|---|---|
| 1: | antes | $\{$ `TRUE` $\}$ |
| | después | $\{\ a = 3\ \}$ |
| 2: | antes | $\{\ a = 3\ \}$ |
| | después | $\{\ a = 3\ \}$ |
| 3: | antes | $\{$ `FALSE` $\}$ |
| | después | $\{$ `FALSE` $\}$ |
| 4: | antes | $\{$ `FALSE` $\}$ |
| | después | $\{$ `FALSE` $\}$ |
| 5: | antes | $\{\ a = 3\ \}$ |
| | después | $\{\ a = 3 \wedge ret = 0\ \}$ |
| 6: | antes | $\{\ a = 3 \wedge ret = 0\ \}$ |
| | después | $\{\ a = 3 \wedge ret = 0 \wedge retValue = ret\ \}$ |

Figura 3.7: Solución obtenida utilizando el nuevo sistema de reglas.

## 3.3. El análisis en programas con procedimientos (métodos)

Cuando se desea implementar un algoritmo de complejidad[1] media o alta, una de las actividades más frecuentes es la fragmentación de dicho algoritmo en diversas partes que encapsulan pequeñas porciones más o menos independientes de su lógica, para luego combinarlas de forma tal de obtener el comportamiento deseado. De esta forma se obtiene una serie de ventajas, como simpleza en la lectura, escritura y comprensión del algoritmo, reutilización de porciones de lógica ya existentes, etc. En este sentido, casi cualquier lenguaje utilizado para construir software en la actualidad provee un mecanismo para aislar lógica en partes más pequeñas y luego combinarlas con el objetivo de implementar algoritmos mucho más complejos. Por esto resulta necesario extender el lenguaje presentado en la sección 3.1 para permitir la definición de dichas partes y su combinación.

Si bien el concepto subyacente tras estas partes que se combinan para implementar algoritmos más complejos es prácticamente el mismo, dichas partes suelen adquirir nombres y características diferentes de acuerdo al contexto en el que existen. Por ejemplo, bajo el paradigma de programación funcional, las partes utilizadas son *funciones*; en el paradigma imperativo suelen ser *procedimientos*, etc. En este trabajo se utilizará el término *método* para referirse a dichas partes, dado que está más estrechamente ligado a la programación con características de orientación a objetos.

### 3.3.1. Lenguaje de programas de números enteros con métodos

A continuación se presenta la adaptación del lenguaje de la sección 3.1 que permite definir y combinar (mediante invocación) *métodos*.

[1] La utilización del término complejidad no hace referencia a la complejidad algorítmica que suele clasificarse en lineal, logarítmica, polinomial, exponencial, etc, sino a lo dificultosa que puede resultar la tarea de definir determinado algoritmo

| | | |
|---|---|---|
| Programa | → | Métodos |
| Métodos | → | Método |
| Métodos | → | Métodos Método |
| Método | → | *nombre* `()` `{` SentenciaEtiquetada `}` |
| | | donde *nombre* es una cadena de caracteres |
| Método | → | *nombre* `(` Parámetros `)` `{` SentenciaEtiquetada `}` |
| Parámetros | → | *variableLocal* |
| Parámetros | → | Parámetros, *variableLocal* |
| ExpresiónZ | → | $z$ con $z \in \mathbf{Z}$ |
| ExpresiónZ | → | *variableLocal* |
| ExpresiónZ | → | - *variableLocal* |
| ExpresiónZ | → | *variableLocal* OperadorZ Operando |
| ExpresiónZ | → | Operando OperadorZ *variableLocal* |
| ExpresiónZ | → | ExpresiónNoLineal |
| ExpresiónZ | → | `invocar` *nombre*`()` |
| ExpresiónZ | → | `invocar` *nombre*`(`Argumentos`)` |
| Argumentos | → | Operando |
| Argumentos | → | Argumentos, Operando |

Las categorías sintácticas faltantes deben ser consideradas exactamente iguales a las del lenguaje de la sección 3.1.

Ahora un programa ha dejado de ser una lista de sentencias numeradas, y se ha transformado en una lista de métodos. A diferencia de las sentencias, el orden de aparición de los métodos no tiene ninguna influencia sobre la semántica del programa. Luego cada método posee: un nombre que permitirá identificarlo al momento de la invocación, una lista de parámetros (posiblemente vacía) mediante los cuales podrá recibir datos provenientes del método invocador y una lista de sentencias numeradas. Las expresiones de números enteros han sido extendidas con la incorporación de la posibilidad de invocar a un método sin argumentos o con argumentos.

Esta modificación, para permitir definiciones e invocaciones a métodos, da lugar a un nuevo, más extenso y más variado conjunto de potenciales errores en la construcción del programa. Nuevamente, se tiene en cuenta que los programas en este lenguaje intermedio son construidos tomando como base a programas escritos en una sintaxis concreta, que hayan superado exitosamente los chequeos sintácticos y semánticos correspondientes. Por lo que se asumirá que los programas a analizar están bien formados y se omitirá un detalle exhaustivo de las condiciones consideradas. De todas maneras, se menciona que se asume que: los métodos que son invocados terminan correctamente, es decir con sentencias '`return` Operando' y no con sentencias '`return`' (la sentencia '`return`' podría aparecer únicamente en el método principal, *main*, del programa); que los métodos son unívocamente identificables porque todos poseen un nombre diferente; que las invocaciones se realizan correctamente, es decir mediante la utilización de nombres de métodos existentes y cantidad de argumentos correspondiente.

### 3.3.2. Análisis de programas de números enteros con métodos

Al analizar un programa interprocedural la mínima unidad a considerar ya no es la que determina la categoría gramatical Programa, sino que será la que determina la categoría gramatical Método. La técnica ya no estará enfocada en analizar programas completos, sino métodos individualmente. De todas formas, en la gran mayoría de los lenguajes de programación existe un nombre de método distinguido, por ejemplo `main`, que se considera el método inicial del programa, por lo tanto analizar el método distinguido tendrá el efecto de analizar el programa completo.

Teniendo en cuenta la diferencia mencionada en el párrafo anterior, es necesario adaptar las definiciones 3.2.1 (Conjunto de nodos del grafo de un programa P), 3.2.2 (Conjunto de ejes del grafo de un programa P) y 3.2.3 (Grafo de un programa P) para que sean *Conjunto de nodos del grafo de un método M*, *Conjunto de ejes del grafo de un método M* y *Grafo de un método M* respectivamente. Estas adaptaciones son directas si se reemplaza la lista de sentencias de un programa monoprocedural generada a partir de la producción "$Programa \rightarrow SentenciaEtiquetada$" por la lista de sentencias de un método de un programa interprocedural generada a partir de la producción "$Mtodo \rightarrow nombre$`() { `$SentenciaEtiquetada$` }`", o la producción "$Mtodo \rightarrow nombre$`(`$Parmetros$`) { `$SentenciaEtiquetada$` }`".

Por otra parte, considerando ahora que un programa es una lista de métodos:

$$P = Mtodo^{+}$$

es necesario introducir las siguientes definiciones.

**Definición 3.3.1 (Nombre del método $M$)** *El nombre de un método $M$ se denota $nombre(M)$ y se define de la siguiente manera:*

$$\begin{array}{lcl} nombre(M) & = & \alpha \text{ si } M = \alpha\texttt{() \{ } SentenciaEtiquetada \texttt{ \}} \\ nombre(M) & = & \alpha \text{ si } M = \alpha\texttt{(}Parmetros\texttt{) \{ } SentenciaEtiquetada \texttt{ \}} \end{array}$$

**Definición 3.3.2 (Sentencias del método $M$)** *La lista de sentencias de un método $M$ se denota $stnc(M)$ y es la secuencia de pares de la forma $(Etq, Stnc)^{+}$ que se define de la siguiente manera:*

$$\begin{array}{lcl} stnc(M) & = & SE_1 \text{ si } M = \alpha\texttt{() \{ } SE_1 \texttt{ \}} \\ stnc(M) & = & SE_1 \text{ si } M = \alpha\texttt{(}Parmetros\texttt{) \{ } SE_1 \texttt{ \}} \end{array}$$

**Definición 3.3.3 (Conjunto de nodos del grafo de llamadas de un programa $P$)** *El conjunto de nodos del grafo de llamada de un programa $P$ se denota $CG^{P}_{nodos}$ y se define de la siguiente manera:*

$$\begin{array}{lcl} CG^{P}_{nodos} & = & \{nombre(M_1)\} \text{ si } P = M_1 \\ CG^{P}_{nodos} & = & \{nombre(M_1)\} \cup CG^{P'}_{nodos} \text{ si } P = M_1 P' \wedge P' \neq [\,] \end{array}$$

**Definición 3.3.4 (Conjunto de ejes del grafo de llamadas de un programa $P$)** *Un eje del grafo de llamadas de un programa $P$ es un par compuesto por dos nombres de*

*métodos no necesariamente diferentes* $(\alpha, \beta)$*. El eje representa la noción de que desde el método* $\alpha$ *se realiza una invocación al método* $\beta$*. El conjunto de ejes del grafo de llamadas de un programa* $P$ *se denota* $CG^P_{ejes}$ *y se define de la siguiente manera:*

$$CG^P_{ejes} = invc(nombre(M_1), stnc(M_1)) \text{ si } P = M_1$$

$$CG^P_{ejes} = invc(nombre(M_1), stnc(M_1)) \cup CG^{P'}_{ejes} \text{ si } P = M_1 P' \wedge P' \neq [\,]$$

*donde* $invc(nombre_1, sentencias_1)$ *es el conjunto de pares donde la primera componente es* $nombre_1$*, o sea el nombre del método invocador, y la segunda componente es el nombre de cada uno de los métodos que son invocados mediante las sentencias de la lista* $sentencias_1$ *que sean del tipo "*$variableLocal$ = `invocar` $nombre$`()`*" o sentencias del tipo "*$variableLocal$ = `invocar` $nombre$`(`$Argumentos$`)`*". Cabe aclarar que por ser un conjunto no hay pares repetidos, o sea, aunque varias sentencias invoquen al mismo método solo existirá un par que representa la relación entre ambos.*

**Definición 3.3.5 (Grafo de llamadas de un programa** $P$**)** *El grafo de llamadas de un programa* $P$*, denotado* $CG^P$ *es el grafo dirigido que contiene como conjunto de nodos a* $CG^P_{nodos}$*, y como conjunto de ejes a* $CG^P_{ejes}$*.*

En la figura 3.9 pude verse el grafo de llamadas del programa del programa de la figura 3.8

```
m0() {                          m2(n, s) {
    (1, a = invocar m1(0));        (1, x = n + s);
    (2, b = invocar m2(1, 2))      (2, return x)
}                               }

m1(k) {
    (1, c = invocar m2(1, 2));
}
```

Figura 3.8: Programa con métodos.

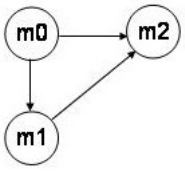


Figura 3.9: Grafo de llamadas correspondiente al programa de la figura 3.8

Una limitación existente en las técnicas de análisis interprocedural que se presentan en esta sección es que, en general, no serán capaces de analizar programas cuyo grafo de llamadas contenga algún ciclo. Es decir, las técnicas no podrán analizar programas donde exista algún tipo de recursión. En la sección 6.1 se presentan algunos indicios de cómo extender las técnicas para que soporten programas con invocaciones recursivas.

Continuando con la adaptación de las definiciones al nuevo lenguaje, es necesario modificar las definiciones de "etiqueta inicial" y "variables del análisis" para que se refieran a *método*, que es la unidad a analizar en programas interprocedurales.

**Definición 3.3.6 (Etiqueta inicial de un método $M$)** *La etiqueta inicial de un método $M$ es la etiqueta de la primera sentencia etiquetada que es considerada cuando el método es ejecutado.*

$$\begin{aligned} Prim(M) &= Etq_1 \text{ si } stnc(M) = (Etq_1, Stnc) \\ Prim(M) &= Etq_1 \text{ si } stnc(M) = (Etq_1, Stnc); M' \end{aligned}$$

**Definición 3.3.7 (Conjunto de variables locales de un método $M$)** *El conjunto de variables locales de un método $M$ se denota $Vars(M)$ y define de la siguiente manera:*

$$Vars(M) = VarsEnStnc(stnc(M))$$

*donde:*

$$\begin{aligned} VarsEnStnc(L) &= \{variableLocal\} \text{ si } L = (Etq, variableLocal\texttt{=}ExpresinZ) \\ VarsEnStnc(L) &= \emptyset \text{ si } L = (Etq, Stnc) \wedge Stnc \neq Asig \\ VarsEnStnc(L) &= VarsEnStnc(L') \\ &\qquad \text{si } L = (Etq, Stnc)L' \wedge Stnc \neq Asig \wedge L' \neq [\,] \\ VarsEnStnc(L) &= \{variableLocal\} \cup VarsEnStnc(L') \\ &\qquad \text{si } L = (Etq, variableLocal\texttt{=}ExpresinZ)L' \wedge L' \neq [\,] \end{aligned}$$

**Definición 3.3.8 (Conjunto de parámetros formales de un método $M$)** *El conjunto de parámetros formales de un método $M$ se denota $Params(M)$ y define de la siguiente manera:*

$$\begin{aligned} Params(M) &= \emptyset \text{ si } M = \alpha\texttt{() \{ } SE_1 \texttt{ \}} \\ Params(M) &= parmetros_1 \text{ si } M = \alpha\texttt{(}parmetros_1\texttt{) \{ } SE_1 \texttt{ \}} \end{aligned}$$

**Definición 3.3.9 (Conjunto de meta-argumentos método $M$)** *El conjunto de meta-argumentos del método $M$, denotado $MetaArgs(M)$, es un conjunto de variables 'frescas', variables que no aparecen en el programa, de la misma cardinalidad que el conjunto de parámetros formales de $M$ ($Params(M)$). Por simpleza se asumirá que el conjunto de meta-argumentos posee variables cuyos nombres son idénticos a cada uno de los nombres de los parámetros formales y contienen el subíndice "$_0$". Por ejemplo, si el conjunto de parámetros formales es $\{a, b\}$, el conjunto de meta-argumentos es $\{a_0, b_0\}$.*

*El conjunto de meta-argumentos es vacío si lo es el conjunto de parámetros formales.*

**Definición 3.3.10 (Conjunto de variables del análisis del método $M$)** *El conjunto de variables del análisis del método $M$ es:*

$$VarsAn(M) = Vars(M) \cup Params(M) \cup MetaArgs(M) \cup \{retValue_{nombre(M)}\}$$

Nuevamente, al igual que el conjunto de variables del análisis de un programa monoprocedural, se asume que la variable $retValue_{nombre(M)}$ tiene un nombre distinguido que no se utiliza en el programa. Además, es necesario que dependa del nombre del método ya que podrá haber tantas variables que representan valores de retorno como métodos en el programa.

Antes de presentar las reglas que permitirán definir el sistema de ecuaciones *Data Flow* del análisis de un programa interprocedural, es necesaria una última definición.

**Definición 3.3.11 (Post-condición del método $M$)** *La post-condición de un método $M$ se denota $PostCond(M)$ y es la disyunción de todas las restricciones obtenidas por el análisis que corresponden al momento después de ejecutar una sentencia de retorno del método.*

$$PostCond(M) = \bigcup_{Etq_i \text{ corresponde a una sentencia } \texttt{return}} Etq_i^{después}$$

Habiendo realizado todas las definiciones necesarias, a continuación se presentan las reglas que resultan comunes a ambas técnicas de análisis interprocedural (simulación de *inlining* y análisis simbólico). Luego, en las sub-secciones correspondientes se desarrollan las particularidades de cada una.

Teniendo en cuenta que el análisis se realiza sobre el método $M$, el conjunto de reglas que relacionan las incógnitas que representan restricciones antes de ejecutar una sentencia, con las restricciones después de ejecutar cada una de las sentencias que la anteceden inmediatamente en el flujo del programa, es el siguiente:

$R_1^{PM}$. $\quad Etq_i^{antes} = D_{universal}^{VarsAn(M)}$ si $Etq_i = Prim(M) \wedge M = \alpha\texttt{() \{ } SE_1 \texttt{ \}}$

$R_2^{PM}$. $\quad Etq_i^{antes} = D_{bindMetaArg} \cup \bigcup_{(j,i) \in G_{ejes}^M} procCond(i, Etq_j)$

si $Etq_i = Prim(M) \wedge M = \alpha\texttt{(}parmetros_1\texttt{) \{ } SE_1 \texttt{ \}}$

$R_3^{PM}$. $\quad Etq_i^{antes} = \bigcup_{(j,i)\in G_{ejes}^M} procCond(i, Etq_j)$ si $Etq_i \neq Prim(M)$

donde:

$D_{bindMetaArg} = nuevoDominio(p_1 = p_{1_0} \wedge p_2 = p_{2_0} \wedge ... \wedge p_k = p_{k_0}, VarsAn(M))$

$Params(M) = \{p_1, p_2, ..., p_k\}$

$MetaArgs(M) = \{p_{1_0}, p_{2_0}, ..., p_{k_0}\}$

Como puede observarse, cuando se trata de la primera sentencia de un método y este no espera argumentos ($R_1^{PM}$), la restricción es el dominio universal, lo que indica que no hay información. Nuevamente, no es necesario realizar la unión con la información que potencialmente proviene de sentencias inmediatamente antecesoras (que realizan saltos a la sentencia $Prim(m)$), porque se estaría realizando la disyunción con el dominio universal, lo cual no aporta información.

Si por el contrario el método sí espera recibir argumentos ($R_2^{PM}$), la restricción es la unión de dos elementos. El primero es el *binding* de las variables que representan parámetros formales con meta-variables que representan valores de los argumentos existentes en una hipotética invocación del método analizado. Este *binding* es útil, por un lado, para obtener especificaciones del método a partir de las restricciones obtenidas (ver sección 1.2.3), y por otro lado, como se verá más adelante, para efectuar el análisis interprocedural en modo simbólico. El segundo elemento es la unión de las restricciones existentes después de cada sentencia del programa que antecede inmediatamente a la primera, utilizándose la versión que incrementa la precisión del análisis considerando las condiciones booleanas de los saltos condicionales. Ahora sí, este segundo elemento es necesario debido a que el sistema de alteración de flujo del lenguaje presentado permitiría ejecutar la primera sentencia más de una vez, y la disyunción no se realizaría con el dominio universal, sino con el *binding* de parámetros y meta-argumentos.

Finalmente, si la sentencia no es la primera del método ($R_3^{PM}$), la regla es igual a la existente en la versión monoprocedural del análisis.

Del conjunto de reglas que relaciona la restricción después de ejecutar una sentencia con la restricción existente antes de ejecutarla, todas las reglas para ambas técnicas interprocedurales son iguales a las presentadas en el análisis de programas mono-procedurales (ver sección 3.2.2) salvo la regla que considera sentencias del tipo "$variableLocal = ExprZ \wedge ExprZ \neq ExprNL''$. En este caso se deben considerar dos situaciones diferentes. Por un lado, si la expresión que se asigna no es una invocación, sí debe utilizarse la regla de la sección 3.2.2 ($R_5^P$). Por el contrario, si la expresión resulta de la invocación de algún método ("`invocar` *nombre*`()`" o "`invocar` *nombre*`(`Argumentos`)`") debe ser tratada de forma diferente.

En este trabajo se considerarán dos maneras de realizar el análisis en presencia de invocaciones, mediante simulación de *inlining* y mediante tratamiento simbólico. En las próximas dos secciones se presentan ambas maneras de analizar invocaciones.

**Análisis interprocedural mediante simulación de *inlining***

El concepto tras este tipo de análisis es el mismo que existe en presencia de *macro-expansiones*. Es decir, cuando se encuentra una invocación a una *macro*, la sentencia que representa dicha invocación es, en algún sentido, reemplazada por el propio cuerpo de la *macro*, pudiendo existir algún tipo de sistema de pasaje de argumentos más bien sintáctico. Además, este reemplazo se efectúa en alguna etapa previa a la ejecución, de forma tal que el hecho de que originalmente haya habido una invocación en el programa, puede pasar desapercibido al momento de ejecutar. En la figura 3.11 se muestra como es posible representar mediante *inlining* la invocación de la sentencia "`2:`" del método *main* de la figura 3.10

La intención es replicar este mismo concepto al momento de analizar una invocación a un método. De esta manera se genera una nueva cantidad de incógnitas en el análisis (al menos dos por cada sentencia del método invocado) y se deben aplicar las reglas mencionadas anteriormente para las sentencias del método invocado que no realizan a su vez nuevas invocaciones. Para las que sí lo hacen, se genera otro nuevo conjunto de incógnitas que se agregan al análisis y se realiza lo mismo sucesivamente hasta encontrar un método que no realice ningún tipo de invocación (al sólo considerar programas cuyo grafo de llamadas contenga ciclos, se asegura dicha condición).

```
main() {                          m2( p1 ) {
    (1, a = 3);                       (1, p1 = p1 + 1);
    (2, ret = invocar m2(a);          (2, b = p1);
    (3, return ret)                   (3, return b);
}                                 }
```

Figura 3.10: Programa interprocedural

```
main() {
    (1, a = 3);
    (2, p1 = a);
    (3, p1 = p1 + 1);
    (4, b = p1);
    (5, ret = b;
    (6, return ret)
}
```

Figura 3.11: Programa que hipotéticamente se consideraría al realizar *inlining* en el método *main* de la figura 3.10

En presencia de una sentencia de la forma:

$$(Etq_i,\ v\ \texttt{= invocar}\ m2\texttt{()})$$

Lo que debe realizarse es lo siguiente:

1. Extender las variables de $Etq_i^{antes}$ para que contengan las variables de $m2$ Es decir:

$$Etq_{precondicion} = Etq_i^{antes} \nearrow VarsAn(m2)$$

En este punto es necesario asumir que $VarsAn(M) \cap VarsAn(m2) = \emptyset$, o sea las variables del método que se está analizando ($M$) son todas diferentes a las del método que invoca ($m2$).

2. Realizar el análisis de $m2$ teniendo en cuenta que:

$$Etq^{antes}_{Prim(m2)} = Etq_{precondicion} \cup \bigcup_{(j,Prim(m2)) \in G^{m2}_{ejes}} procCond(Prim(m2), Etq_j)$$

Durante el análisis de $m2$ se deben utilizar las mismas reglas que se utilizan durante el análisis de $M$ (el método principal del análisis). Es posible que $m2$ realice invocaciones a otros métodos, y en este caso simplemente se cambia el rol, y $m2$ pasa a ser el método invocador. El hecho de limitar el análisis a programas cuyo grafo de llamadas no contenga ciclos, asegura que finalmente se encontrará un método en el que no haya invocaciones y no deban realizarse las acciones que aquí se describen.

3. Finalmente, una vez construido el sistema de ecuaciones correspondientes al análisis de $m2$, se debe reflejar la asignación del resultado de la invocación. Para esto se obtiene la post-condición de $m2$, se realiza el *binding* del valor de retorno con la variable que se está definiendo en $M$, y luego se eliminan de la restricción las variables del análisis de $m2$.

$$Etq_i^{después} = (PostCond(m2) \cap D_{bindRet}) \searrow varsAn(m2)$$

donde:

$$D_{bindRet} = nuevoDominio(v = retValue_{m2}, VarsAn(M) \cup VarsAn(m2))$$

Notar que es necesario eliminar las variables del análisis de $m2$ para que, en la restricción existente en el punto del programa después de la invocación y las subsiguientes, sólo se predique sobre las variables del método llamador, y no se tenga información sobre las variables asociadas al método invocado.

Si la sentencia a analizar es:

$$(Etq_i,\ v \texttt{ = invocar } m2(arg_1, arg_2, ..., arg_k))$$

o sea una invocación con argumentos, debe realizarse lo siguiente:

1. Extender las variables de $Etq_i^{antes}$ para que contengan las variables de $m2$, de la misma forma que se hace en presencia de una invocación sin argumentos, además realizar el *binding* de los argumentos de la invocación con los parámetros formales.

$$Etq_{precondicion} = (Etq_i^{antes} \nearrow VarsAn(m2)) \cap D_{bindArgs}$$

donde:

$$D_{bindArgs} = nuevoDominio(p_1 = arg_1 \wedge p_2 = arg_2 \wedge ... \wedge p_k = arg_k, \\ VarsAn(M) \cup VarsAn(m2))$$

Finalmente considerar los items 2. y 3. de la descripción de pasos a realizar cuando se considera una invocación sin argumentos.

Una cuestión muy importante a considerar es que esta técnica de análisis interprocedural es únicamente válida cuando la semántica del pasaje de parámetros del lenguaje es "por copia". Si se analiza el método *main* del programa de la figura 3.10, la restricción después de ejecutar la sentencia "2" del método main es $\{ a = 3 \wedge ret = 4 \}$, lo que refleja que la variable $a$ no es modificada por el método $m2$.

Esta característica se debe a que al momento de iniciar el análisis del método invocado existe el *binding* entre los parámetros formales y los argumentos reales, pero las operaciones siempre se realizan sobre los parámetros formales, entonces los argumentos reales son inmutables en el *scope* del método invocado.

Este tipo de pasaje de parámetros es el utilizado, por ejemplo, en el lenguaje de programación Java, lo que hace que, al menos en lo que respecta al pasaje de parámetros, la técnica sea compatible con dicho lenguaje.

### Análisis interprocedural simbólico

Si un método $m1$ realiza, por ejemplo, cien invocaciones a otro método $m2$ en cien sentencias diferentes y el análisis interprocedural se realiza mediante simulación de *inlining*, el método $m2$ será analizado las cien veces, generando cien conjuntos de nuevas incógnitas al análisis y teniendo que aplicar cien veces las reglas para las mimas sentencias, dando lugar a cien veces la cantidad de ecuaciones del método invocado. Esta situación se presenta aun cuando las invocaciones se realizan utilizando los mismos argumentos. La idea tras el análisis interprocedural simbólico es evitar esta situación y realizar el análisis de cada método involucrado de forma aislada y sólo una única vez utilizando valores simbólicos para los argumentos. Luego se realizará el *binding* de los valores simbólicos con los argumentos, al momento de realizarse la invocación.

Si se quiere analizar el método $M$ es necesario analizar previamente todos los métodos que son invocados directa o indirectamente a partir de $M$. Sabiendo que el análisis sólo puede realizarse en programas cuyo grafo de llamadas no contiene ciclos, es posible considerar como sub-grafo del grafo de llamadas del programa el árbol cuya raíz es $M$. Luego el análisis debe realizarse de la siguiente manera:

1. analizar los métodos que corresponden a hojas del árbol mencionado anteriormente. Estos métodos, por ser hojas, no poseen ninguna invocación y pueden ser analizados de la misma forma que con *inlining*, o sea realizando el *binding* de los parámetros formales con meta-argumentos cuando corresponda.

2. realizar progresivamente el análisis de los métodos del árbol en cuestión cuyos hijos ya estén todos analizados. En la primera iteración sólo podrán analizarse los métodos cuyos hijos son todos raíces. En la segunda iteración podrán analizarse los método que no se hayan analizado en la primera iteración cuyos hijos sean raíces o ya hayan sido analizados, y así sucesivamente hasta poder analizar finalmente el método $M$. En esta etapa las invocaciones a métodos deben ser analizadas de la forma que se explica más adelante.

Finalmente el análisis interprocedural simbólico en presencia de una invocación sin

argumentos, como por ejemplo:

$$(Etq_i,\ v\ \texttt{= invocar}\ m2\texttt{()})$$

se realiza de la siguiente manera:

1. Eliminar de $PostCond(m2)$ las variables locales para que el valor de retorno del método quede representado mediante una constante. Si el método no tiene parámetros se asume que retorna un valor constante. En caso de que el valor de retorno del método no pueda ser representado mediante una expresión lineal la variable de retorno no tendrá restricción y el resultado de dicha operación será el dominio universal.

$$D_2 = PostCond(m2) \searrow Vars(m2)$$

2. Extender $D_2$ y $Etq_i^{antes}$ para que tengan el mismo conjunto de variables. A $D_2$ se le deben agregar las variables locales, los parámetros, los meta-argumentos y la variable de retorno de $M$. A $Etq_i^{antes}$ debe agregársele $retValue_{m2}$.

$$D_3 = D_2 \nearrow VarsAn(M)$$

$$D_4 = Etq_i^{antes} \nearrow \{retValue_{m2}\}$$

3. Intersecar $D_3$ y $D_4$ y realizar el *binding* del valor de retorno de $m2$ con la variable que se define $v$, para reflejar el resultado de la invocación.

$$D_5 = (D_3 \cap D_4) \cap D_{bindRet}$$

$$D_{bindRet} = nuevoDominio(v = retValue_{m2}, VarsAn(M) \cup \{retValue_{m2}\})$$

4. Eliminar $retValue_{m2}$ para que toda la restricción predique únicamente sobre $varsAn(M)$

$$Etq_i^{después} = D_5 \searrow \{retValue_{m2}\}$$

Cabe aclarar que la expresión anterior constituye la regla (o parte de la función de transferencia) a aplicar para las sentencias que realizan invocaciones sin argumentos. Se ha llegado a la expresión en forma de procedimiento para facilitar su notación y comprensión, pero debe quedar claro que se trata de una única regla.

Si la invocación sí tiene argumentos, como por ejemplo:

$$(Etq_i,\ v\ \texttt{= invocar}\ m2\texttt{(}arg_1, arg_2, ..., arg_k\texttt{)})$$

debe realizarse lo siguiente:

1. Realizar el binding de los argumentos con los meta-argumentos en la post-condición del método invocado. Se supone que los métodos invocados ya han sido analizados, así que es posible obtener su post-condición.

$$D_1 = (PostCond(m2) \nearrow \{arg_1, arg_2, ..., arg_k\}) \cap D_{bind}$$

$$D_{bind} = nuevoDominio( \quad ma_1 = arg_1 \wedge ma_2 = arg_2 \wedge ... \wedge ma_k = arg_k, \\ VarsAn(m2) \cup \{arg_1, arg_2, ..., arg_k\})$$

donde:

$$MetaArgs(m2) = \{ma_1, ma_2, ..., ma_k\}$$

2. Eliminar de $D_1$ las variables locales, los parámetros formales y los meta-argumentos para que el valor de retorno del método quede sólo en función de los argumentos.

$$D_2 = D_1 \searrow (Vars(m2) \cup Params(m2) \cup MetaArgs(m2))$$

3. Extender $D_2$ y $Etq_i^{antes}$ para que tengan el mismo conjunto de variables. A $D_2$ se le deben agregar las variables locales y los parámetros de $M$ que no sean argumentos de la invocación, más los meta-argumentos y la variable de retorno del método invocador. A $Etq_i^{antes}$ debe agregársele $retValue_{m2}$.

$$D_3 = D_2 \nearrow (VarsAn(M) - \{arg_1, arg_2, ..., arg_k\})$$

$$D_4 = Etq_i^{antes} \nearrow \{retValue_{m2}\}$$

4. Intersecar $D_3$ y $D_4$ y realizar el *binding* del valor de retorno de $m2$ con la variable que se define $v$, para reflejar el resultado de la invocación.

$$D_5 = (D_3 \cap D_4) \cap D_{bindRet}$$

$$D_{bindRet} = nuevoDominio(v = retValue_{m2}, VarsAn(M) \cup \{retValue_{m2}\})$$

5. Eliminar $retValue_{m2}$ para que toda la restricción predique únicamente sobre $VarsAn(M)$

$$Etq_i^{después} = D_5 \searrow \{retValue_{m2}\}$$

Al igual que en el caso de las invocaciones sin argumentos, la expresión anterior es la regla (o parte de la función de transferencia) a aplicar en las sentencias que realizan invocaciones con argumentos. Se ha llegado a la expresión en forma de procedimiento para facilitar su notación y comprensión, pero debe quedar claro que también se trata de una única regla.

## 3.4. Solución del sistema de ecuaciones de *Data Flow*

En la sección 3.2.2 se muestra como a partir del código fuente de un programa de $n$ sentencias pueden obtenerse $2n$ ecuaciones de $2n$ incógnitas. Cada una de las incógnitas representa una restricción lineal y la solución al sistema de ecuaciones es el resultado del análisis de restricciones lineales.

Una de las maneras de resolver el sistema de ecuaciones es utilizando la noción de punto fijo. Para esto es útil reescribir las ecuaciones como se describe a continuación.

El conjunto de incógnitas se representa mediante una $2n$-upla de elementos:

$$\overrightarrow{RL} = \{Etq_1^{antes}, Etq_1^{después}, Etq_2^{antes}, Etq_2^{después}, ..., Etq_n^{antes}, Etq_n^{después}\}$$

Además puede considerarse que las ecuaciones definen una función $F$ tal que:

$$\overrightarrow{RL} = F(\overrightarrow{RL})$$

$$F : (Dominio)^{2n} \rightarrow (Dominio)^{2n}$$

o sea $F$ toma una 2n-upla de dominios y retorna otra 2n-upla de dominios.

donde más específicamente:

$$F(\overrightarrow{RL}) = (F_{Etq_1^{antes}}(\overrightarrow{RL}), F_{Etq_1^{después}}(\overrightarrow{RL}), F_{Etq_2^{antes}}(\overrightarrow{RL}), ..., F_{Etq_n^{después}}(\overrightarrow{RL}))$$

Por ejemplo, considerando el sistema de ecuaciones de la figura 3.5 se tiene:

$$F_{Etq_1^{antes}}(\overrightarrow{RL}) = D_{universal}^{\{a,\ ret,\ retValue\}}$$

o sea la función constante que retorna siempre $D_{universal}^{\{a,\ ret,\ retValue\}}$

y

$$F_{Etq_6^{antes}}(\overrightarrow{RL}) = Etq_2^{después} \cup Etq_5^{después}$$

o sea la función que devuelve la disyunción de la cuarta y décima componente de la 12-upla $\overrightarrow{RL}$.

Representado el sistema de ecuaciones mediante la relación $\overrightarrow{RL} = F(\overrightarrow{RL})$ se tiene que un punto fijo de la función $F$ es una solución al sistema de ecuaciones y en consecuencia es el resultado del análisis.

Una de las características principales de la función $F$ es que es monótona. Es decir, si se considera que el conjunto de $(Dominio)^{2n}$, o sea el conjunto de 2n-uplas de dominios, es un conjunto parcialmente ordenado (ver sección 2.2)utilizando la relación:

$$\overrightarrow{RL} \sqsubseteq \overrightarrow{RL}' \Leftrightarrow \forall i : 1 \leq i \leq 2n \Rightarrow \overrightarrow{RL}_i \subseteq \overrightarrow{RL}'_i$$

luego puede demostrarse (ver A.0.1) que

$$\overrightarrow{RL} \sqsubseteq \overrightarrow{RL}' \Rightarrow F(\overrightarrow{RL}) \sqsubseteq F(\overrightarrow{RL}')$$

Teniendo en cuenta que la función $F$ es monótona y que el conjunto de dominios de poliedros convexos cerrados es un reticulado completo (ver 2.2) es posible enmarcar el análisis de restricciones lineales dentro del *Monotone Framework*.

El *Monotone Framework* es un marco teórico dentro del cual puede encontrarse una gran cantidad de análisis estáticos de código como, por ejemplo, análisis de expresiones disponibles (*Available Expressions*), análisis de definiciones alcanzables (*Reaching Definitions*), análisis de expresiones frecuentes (*Very Busy Expressions*), análisis de variables vivas (*Live Variables*), entre otros. El *Monotone Framework* es una forma de agrupar las similitudes de los análisis de programas y permitir razonar de forma general y abstracta. De esta forma, por ejemplo, se presentan varios métodos de resolución de ecuaciones de *Data Flow* para los análisis que se ajusten al mencionado *framework*.

Una instancia del *Monotone Framework* está compuesta de un conjunto $L$ que sea un reticulado completo, la función $F$ que describe las ecuaciones de *Data Flow* del análisis, el grafo $G$ que describe las alternativas de flujo entre las sentencias del programa, un conjunto $E$ de etiquetas que indican el inicio del análisis y el valor $\iota$ asociado a dichas etiquetas. Una descripción detallada del *Monotone Framework* puede encontarse en [NNH99].

El análisis de restricciones lineales se ajusta al *Monotone Framework* de la siguiente manera. $L$ es el conjunto de dominios de poliedros convexos cerrados (ver sección 2.2), $F$ es la función que se construye a partir de las ecuaciones de *Data Flow* que surgen de las reglas presentadas en la sección 3.2.2. $G$ es $G^P$ (ver definición 3.2.3). $E$ es el conjunto con un único elemento $Prim(P)$ (ver definición 3.2.5). Finalmente, el valor $\iota$ es $D_{universal}$ indicando que no hay información al inicio del análisis.

En el *Monotone Framework* existen varias aproximaciones para obtener el resultado del análisis. En este trabajo se considerará la llamada MFP [2]. El algoritmo que calcula la solución MFP al sistema de ecuaciones de *Data Flow* se muestra en la figura 3.12

El algoritmo toma como entrada una instancia del *Monotone Framework*, y usa un *array*, *Análisis*, para almacenar $MPF^{antes}$, o sea la lista de restricciones antes de cada una de las sentencias del programa. Dicho *array* es indexado utilizando las etiquetas de cada sentencia del programa. El algoritmo también utiliza una *worklist* (lista de tareas) $W$ donde se encuentran ejes del grafo de un programa. La presencia de un eje en $W$ indica que el análisis ha cambiado para la sentencia referida por la primera componente del par. Finalmente el algoritmo retorna el resultado en dos listas que corresponden a las restricciones antes y después de cada una de las sentencias del programa.

Si bien es posible utilizar el método de resolución de ecuaciones de *Data Flow* MFP tal como ha sido presentado, en la práctica pueden encontrase algunos inconvenientes.

[2] MFP son las siglas de Maximal Fixed Point (punto fijo maximal), aunque en realidad el algoritmo computa el menor punto fijo de acuerdo a la relación $\sqsubseteq$. El nombre se conserva en concordancia con la literatura clásica

ENTRADA: una instancia del *Monotone Framework* $(L, F, G, E, \iota)$

SALIDA: $MPF^{antes}, MPF^{después}$

ALGORITMO: Paso 1: Inicialización de $W$ y *Análisis*
$W$ := nil;
para cada $(Etq, Etq')$ en $G$ hacer;
$W := cons((Etq, Etq'), W)$;
para cada $Etq$ en $G$ hacer;
si $Etq \in E$ entonces $Análisis[Etq] = \iota$
sino $Análisis[Etq] = \bot_L$

Paso 2: Iteración (actualización de $W$ y *Análisis*)
mientras $W \neq nil$ hacer
$Etq$ := fst(head($W$));
$Etq'$ := snd(head($W$));
$W$ := tail($W$);
si $F_{Etq}(Análisis[Etq]) \not\sqsubseteq Análisis[Etq']$ entonces
$Análisis[Etq'] := Análisis[Etq'] \sqcup F_{Etq}(Análisis[Etq])$
para cada $Etq''$ tal que $(Etq', Etq'')$ esta en $G$ hacer
$W := cons((Etq', Etq''), W)$;

Paso 3: Presentación del resultado ($MPF^{antes}$ y $MPF^{después}$)
para cada $Etq$ en $G$ hacer
$MPF^{antes}_{Etq} := Análisis[Etq]$;
$MPF^{después}_{Etq} := F_{Etq}(Análisis[Etq])$;

Figura 3.12: Algoritmo MFP para solucionar ecuaciones de *Data Flow*.

Por ejemplo, si se utiliza MFP para resolver el sistema de ecuaciones que se obtiene de aplicar las reglas al programa de la figura 3.13 ocurre lo siguiente. La primera vez que se considera el eje $(4, 3)$ se tiene que:

$Análisis[4] = \{x = 0\}$ y $Análisis[3] = \bot_l = D_{vacio}$

entonces

$F_4(\{x = 0\}) = \{x = 0\}$ y como $\{x = 0\} \not\subseteq D_{vacio}$ entonces $Análisis[3] = \{x = 0\}$.

Por haberse modificado $Análisis[3]$, el eje $(3, 4)$ se agrega a la *worklist*. Cuando se considere el eje $(3, 4)$ se tiene que:

$Análisis[3] = \{x = 0\}$ y $Análisis[4] = \{x = 0\}$

entonces

$F_3(\{x = 0\}) = \{x = 1\}$ y como $\{x = 1\} \not\subseteq \{x = 0\}$ entonces *Análisis*[4] $= \{x = 0\} \cup \{x = 1\}$.

Esta vez, por haberse modificado *Análisis*[4], el eje $(4, 3)$ se vuelve a agregar a la *worklist*.

```
(1, x = 0);
(2, goto 4);
(3, x = x + 1);
(4, if (x < 1000000) goto 3)
(5, return x);

    (a)
```

```
int x = 0;
while (x < 1000000) {
    x = x + 1;
}
return x;

    (b)
```

Figura 3.13: (a) Programa que itera un millón de veces en la sintaxis del análisis (b) versión del programa en utilizando una hipotética sintaxis concreta

Es claro que este proceso se repetirá tantas veces como indique la guarda del ciclo; en este caso, un millón.

Una situación aun peor se presenta cuando la guarda depende de algún valor simbólico como, por ejemplo, un parámetro formal $(x < param_1)$. En este caso, debido a que la semántica del programa está representada con dominios de poliedros convexos cerrados, el valor de $param_1$ es un número entero que hipotéticamente puede ser tan grande como se quiera, con lo cual el análisis debe considerar todos los casos posibles que son infinitos, el algoritmo nunca se detendría y se llegaría a un *overflow* por alcanzar el valor máximo estipulado por la representación numérica de los enteros en una computadora, o alguna anomalía similar.

Situaciones como las descriptas hacen que en la práctica no sea razonable utilizar el algoritmo tal como se lo presentó, y para solucionar esos inconvenientes se realizan aproximaciones al punto fijo utilizando el operador *widening*.

### 3.4.1. *Widening*

**Definición 3.4.1 (Operador de cota superior)** *Un operador* $\tilde{\bigsqcup} : L \times L \to L$ *sobre un reticulado completo* $L$ *es un operador de cota superior si y sólo sí:*

$$l_1 \sqsubseteq (l_1 \tilde{\bigsqcup} l_2) \wedge l_2 \sqsubseteq (l_1 \tilde{\bigsqcup} l_2)$$

**Definición 3.4.2 (Operador de *widening*)** *Un operador* $\nabla : L \times L \to L$ *sobre un reticulado completo* $L$ *es un operador de* widening *si y sólo si:*

- *es un operador de cota superior*
- *para todas las cadenas ascendentes* $(l_n)_n$*, la cadena* $(l_n^{\nabla})_n$ *eventualmente se estabiliza.*

La idea tras el operador de *widening* es la siguiente. Si se tiene una función monótona sobre un reticulado completo, como la que se deriva del sistema de ecuaciones de *Data Flow*, se puede calcular la secuencia $(f_\nabla^n)$ definida de la siguiente manera:

$$f_\nabla^n = \begin{cases} \bot & \text{si } n = 0 \\ f_\nabla^{n-1} & \text{si } n > 0 \wedge f(f_\nabla^{n-1}) \sqsubseteq f_\nabla^{n-1} \\ f_\nabla^{n-1} \nabla f(f_\nabla^{n-1}) & \text{en otro caso} \end{cases}$$

El operador de widening asegura que la cadena ascendente se estabiliza, o sea, en algún momento se cumple la relación $f(f_\nabla^{n-1}) \sqsubseteq f_\nabla^{n-1}$. Y finalmente una serie de resultados que pueden encontrarse en la sección 4.2 de [NNH99] asegura que $lfp(f) \sqsubseteq f_\nabla^n$, es decir, el resultado que se obtiene de calcular la cadena ascendente $f_\nabla^n$ incluye al resultado del algoritmo exacto de punto fijo de la figura 3.12.

La conclusión es que se sacrifica precisión en el análisis ya que en general $lfp(f) \sqsubset f_\nabla^n \wedge lfp(f) \neq f_\nabla^n$, pero se solucionan los problemas relacionados al algoritmo MFP. Además en la práctica unas pocas iteraciones suelen ser suficientes para que la cadena ascendente se estabilice, con lo cual la diferencia en tiempo de cómputo es muy significativa. Estas características hacen que en la gran mayoría de los análisis estáticos de código se utilice esta aproximación utilizando algún operador de *widening* particular.

La decisión de qué operador de *widening* utilizar influirá directamente en el balance entre precisión y performance. Por ejemplo, podría demostrarse que el operador $\nabla_{True}$ definido de la siguiente forma:

$$l_1 \nabla_{True}\ l_2 = D_{universal}$$

es un operador de widening. Por un lado es claro que computar el resultado de aplicar el operador resulta muy poco costoso, además se podría conjeturar que logra estabilizar la cadena ascendente en muy pocas iteraciones. Sin embargo, el operador hace que se pierda toda la información contenida en sus operandos, lo que lo hace muy poco útil.

En este trabajo se utilizará el operador de *widening* que se obtiene se relacionar los trabajos [BHZ04], [BGP99] y [BHRZ03].

En [BHZ04] se define el siguiente operador de *widening* que puede ser aplicado a dominios de poliedros convexos cerrados:

$$D_1 \nabla_{EM} D_2 = h^\nabla(D_1, D_2')$$

donde:

$$D_2' = \begin{cases} D_2 & \text{si } D_1 \vdash_{EM} D_2 \\ \uplus(D_1 \cup D_2) & \text{en otro caso} \end{cases}$$

La relación $D_1 \vdash_{EM} D_2$ se cumple si y solo si $D_1 = D_{vacio} \vee (D_1 \vdash_P D_2 \wedge \forall p_2 \in D_2 : \exists p_1 \in D_1 : d_1 \subseteq d_2)$. Donde $D_1 \vdash_P D_2$ si y solo si $\forall p_1 \in D_1 : \exists p_2 \in D_2 : d_1 \subseteq d_2)$

En [BHZ04] no se define el operador $h^\nabla$, sino que se describen las propiedades que debe cumplir. En [BGP99] sí es posible encontrar una instancia para el operador que se define de la siguiente forma:

ENTRADA: $D_1 = \{d_{1_1}, d_{1_2}, ..., d_{1_m}\}$, $D_2 = \{d_{2_1}, d_{2_2}, ..., d_{2_n}\}$
SALIDA: $P = D_1 \nabla D_2$

$P := D_{vacio}$;
$marca[1..n] := false$;
para cada $1 \leq i \leq n \wedge 1 \leq j \leq m$ hacer
    si $d_{1_j} \subseteq d_{2_i}$ entonces
        $P := P \vee d_{1_j} \hat{\nabla} d_{2_i}$
        $marca[i] := true$;
para cada $1 \leq i \leq n$ hacer
    si $marca[i] = false$ entonces
        $P := P \vee d_{2_i}$

Nuevamente, en la definición $h^{\nabla}$ se deja abierta la posibilidad de utilizar cualquier operador de *widening* ($\hat{\nabla}$). La diferencia radica en que dicho operador debe ser para poliedros y no para dominios.

En [BHRZ03] se presenta la formalización de un operador para poliedros convexos cerrados llamado de *standard widening* cuya idea fue introducida en [Hal79].

$$P_1^{vars} \nabla P_2^{vars} = \begin{cases} P_2^{vars} & \text{si } P_1^{vars} = P_{vacio}^{vars} \\ nuevoPoliedro(C_1' \cup C_2', vars) & \text{en otro caso} \end{cases}$$

donde:

$P_i^{vars} = nuevoPoliedro(C_i, vars)$

$C_1' = \{\beta \in repr_{\geq}(C_1) | P_2 \subseteq nuevoPoliedro(\{\beta\}, vars)\}$

$C_2' = \{\gamma \in repr_{\geq}(C_2) | \exists \beta \in repr_{\geq}(C_1) : P_1 = nuevoPoliedro((repr_{\geq}(C_1) \backslash \{\beta\}) \cup \{\gamma\})\}$

$$\begin{aligned} repr_{\geq}(C) = &\{\langle -a, x\rangle \geq -b | (\langle a, x\rangle = b) \in C\} \cup \\ &\{\langle a, x\rangle \geq b | (\langle a, x\rangle \bowtie b) \in C, \bowtie \in \{\geq, =\}\} \end{aligned}$$

Básicamente la idea de este operador de *widening* es que el resultado de $P_1 \nabla P_2$ es el poliedro compuesto por todas las restricciones de $P_1$ que son conservadas por $P_2$. Para mejorar la precisión del operador las restricciones de la forma $\alpha = \beta$ se representan mediante la disyunción de restricciones $\alpha \geq \beta \wedge \alpha \leq \beta$, de esta forma aunque $P_2$ no conserve la restricción $\alpha = \beta$ aun puede conservar alguna de las dos desigualdades.

## 3.5. Linealización de relaciones no lineales

Tomando las ideas presentadas en [Cla96] es posible lograr que algunas expresiones que no son naturalmente lineales, y deberían ser consideradas por el análisis como expre-

siones de la categoría sintáctica ExpresiónNoLineal, puedan igualmente aportar información a las restricciones lineales.

En presencia de sentencias de la forma "`var1 = var2 / cte`" se puede inferir la relación lineal "$var2 \geq cte \times var1 \wedge var2 < cte \times (var1 + 1)$".

En presencia de sentencias de la forma "`var1 = var2 % cte`", donde el símbolo `%` debe ser interpretado como el operador módulo, se puede inferir la relación lineal "$var2 \geq cte \times tmp \wedge var2 < cte \times (tmp + 1) \wedge var1 = var2 - cte \times tmp$", donde $tmp$ es una nueva variable libre que se agrega a la restricción.

# Capítulo 4

# Implementación y resultados obtenidos

En este capítulo se describe la implementación de las tres técnicas presentadas en el capítulo 3, análisis mono-procedural de programas que realizan operaciones con números enteros, análisis interprocedural mediante simulación de *inlining* y análisis interprocedural simbólico. También se mostrarán y analizarán los resultados obtenidos a partir de la implementación realizada.

## 4.1. Implementación

Las técnicas presentadas en este trabajo fueron implementadas para operar sobre un subconjunto del lenguaje de programación Java. Dicho subconjunto consiste básicamente de programas cuyos métodos realizan únicamente operaciones con números enteros, sin, por ejemplo, instanciar nuevos objetos.

En la figura 4.1 se muestra un diagrama que contiene los principales componentes que intervienen en la aplicación.

Se parte de un programa escrito en lenguaje Java. Luego se lo compila utilizando un compilador Java, y se obtiene el *bytecode*. En esta etapa se realizan las verificaciones semánticas que la técnica ya asume realizadas. Luego el *bytecode* del programa se introduce en el analizador para que sean descubiertas las restricciones correspondientes. Finalmente el analizador genera un archivo conteniendo las restricciones descubiertas.

El analizador está basado en el *framework* de optimización para Java llamado *Soot* [VRHS+99]. *Soot* provee una representación intermedia llamada *Jimple* [**?**]. El lenguaje presentado en el capítulo 3 es básicamente un subconjunto de *Jimple*. El analizador recibe el *bytecode* y utiliza el "conversor" a *Jimple* provisto por *Soot*. Luego se utiliza el *framework* de análisis de *Data Flow* también provisto por *Soot* para derivar las ecuaciones y resolverlas.

Para representar los dominios de poliedros convexos cerrados y sus operaciones se utilizó *PolyLib* [Loe99]. Debido a que *PolyLib* está implementada en el lenguaje de programación C, fue necesario utilizar *JNI* (*Java Native Interface*) para comunicar el análisis *Data Flow*, implementado en Java, con *PolyLib*.

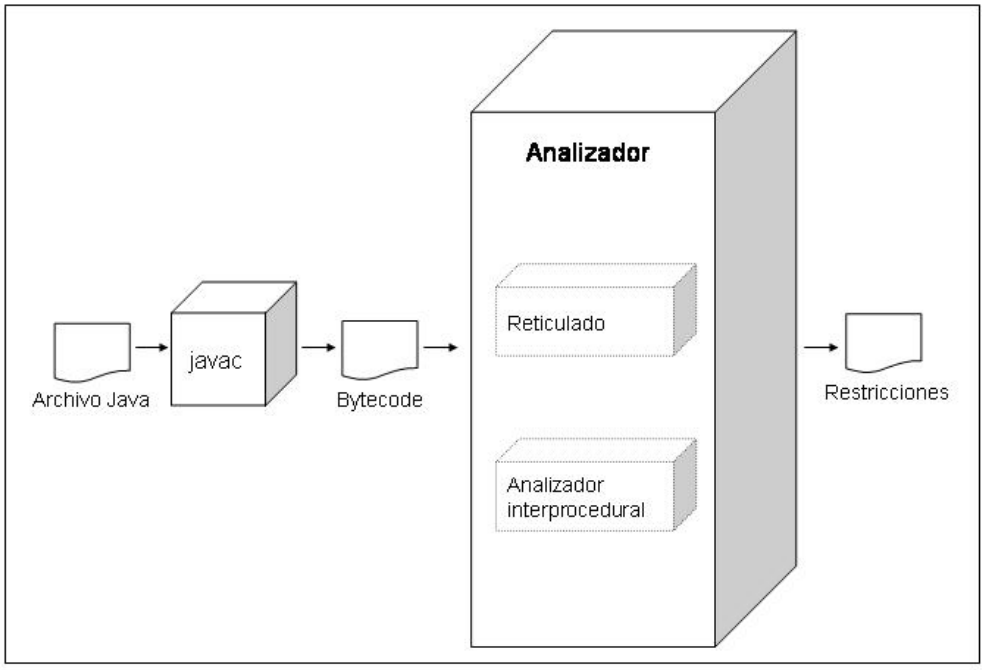


Figura 4.1: Diagrama de la implementación de la técnica.

Un aspecto interesante de la implementación es que, tal como se ve en el diagrama de la figura 4.1, hay dos componentes distinguidos dentro del analizador. Los mismos fueron implementados como componentes intercambiables o *plug-ins*. El primero de ellos es el reticulado que se utiliza para realizar el análisis *Data Flow* enmarcado en el *Monotone Framework*. Para implementar las técnicas del capítulo 3 sólo es necesaria una única implementación de reticulado que es la correspondiente a dominios de poliedros convexos cerrados. Pero, como se verá en el capítulo 5, es necesario extender el reticulado para implementar las ideas que allí se presentan. Por esto resulta útil que el componente sea diseñado como *plug-in*. El segundo componente que fue implementado como *plug-in* es el que realiza el análisis interprocedural. En este caso, existen dos implementaciones para el análisis del programas de números enteros con métodos, que corresponden al análisis interprocedural mediante simulación de *inlining* y al análisis interprocedural simbólico.

El mecanismo de *plug-ins* fue implementado utilizando interfaces. En el apéndice B, se muestra la información relevante de las interfaces involucradas.

## 4.2. Resultados obtenidos

En esta sección se presentan los resultados obtenidos al ejecutar la implementación del descubridor de relaciones lineales entre variables.

### 4.2.1. Ejemplo 1

Como primer ejemplo de los resultados obtenidos con la implementación de la técnica de análisis mono-procedural, se puede mencionar cada una de las restricciones de las figuras: 1.4, 1.5, 2.1 y 3.7

### 4.2.2. Ejemplo 2

En este ejemplo se analizará lo que ocurre en programas sencillos que únicamente incrementan un acumulador.

```
    public int acum1(int n) {
        int ac = 0;
        for (int i = 0; i < n; i++){
1:          ac = ac + 1;
        }
2:      return ac;
    }
```

(a)

```
    public int acum2(int n) {
        int ac = 0;
        for (int i = 0; i < n; i++){
1:        ac = ac + 2;
        }
2:      return ac;
    }
```

(b)

```
    public int acum3(int n) {
        int ac = 0;
        int k = 3;
        for (int i = 0; i < n; i++){
1:          ac = ac + k;
        }
2:      return ac;
    }
```

(c)

```
    public int acumk(int n, int k) {
        int ac = 0;
        for (int i = 0; i < n; i++){
1:          ac = ac + k;
        }
2:      return ac;
    }
```

(d)

Las restricciones obtenidas son:

| | | |
|---|---|---|
| (a) | $1^{antes}$ | $\{ n = n_0 \wedge ac = 0 \wedge i = 0 \wedge n_0 \geq 1 \} \cup \{ n = n_0 \wedge ac = i \wedge i \geq 1 \wedge n_0 \geq i + 1 \}$ |
| | $1^{después}$ | $\{ n = n_0 \wedge ac = i + 1 \wedge i \geq 0 \wedge n_0 \geq i + 1 \}$ |
| | $2^{antes}$ | $\{ n = n_0 \wedge ac = 0 \wedge n_0 \leq 0 \} \cup \{ n = n_0 \wedge ac = n \wedge n \geq 1 \}$ |
| (b) | $1^{antes}$ | $\{ n = n_0 \wedge ac = 0 \wedge i = 0 \wedge n_0 \geq 1 \} \cup \{ n = n_0 \wedge ac = 2i \wedge i \geq 1 \wedge n_0 \geq i + 1 \}$ |
| | $1^{después}$ | $\{ n = n_0 \wedge ac = 2i + 2 \wedge i \geq 0 \wedge n_0 \geq i + 1 \}$ |
| | $2^{antes}$ | $\{ n = n_0 \wedge ac = 0 \wedge n_0 \leq 0 \} \cup \{ n = n_0 \wedge ac = 2n \wedge n \geq 1 \}$ |
| (c) | $1^{antes}$ | $\{n = n_0 \wedge ac = 0 \wedge k = 3 \wedge i = 0 \wedge n_0 \geq 1\} \cup$ $\{n = n_0 \wedge ac = 3i \wedge k = 3 \wedge i \geq 1 \wedge n_0 \geq i + 1\}$ |
| | $1^{después}$ | $\{ n = n_0 \wedge ac = 3i + 3 \wedge k = 3 \wedge i \geq 0 \wedge n_0 \geq i + 1 \}$ |
| | $2^{antes}$ | $\{ n = n_0 \wedge ac = 0 \wedge k = 3 \wedge n_0 \leq 0 \} \cup \{ n = n_0 \wedge ac = 3n \wedge k = 3 \wedge n \geq 1 \}$ |
| (d) | $1^{antes}$ | $\{n = n_0 \wedge k = k_0 \wedge ac = 0 \wedge i = 0 \wedge n_0 \geq 1\} \cup$ $\{n = n_0 \wedge k = k_0 \wedge i \geq 1 \wedge n_0 \geq i + 1\}$ |
| | $1^{después}$ | $\{ n = n_0 \wedge k = k_0 \wedge i \geq 0 \wedge n_0 \geq i + 1 \}$ |
| | $2^{antes}$ | $\{ n = n_0 \wedge k = k_0 \wedge ac = 0 \wedge n_0 \leq 0 \} \cup \{ n = n_0 \wedge k = k_0 \wedge n \geq 1 \}$ |

Es interesante observar como en el punto $2^{antes}$ de los programas (a), (b) y (c) se logra obtener el valor exacto de la variable *ac*. Cabe destacar que aunque en el programa (c) el

valor de $ac$ queda determinado por la relación $ac = k \times n$, esta relación sí es lineal debido a que se detecta que la variable $k$ es constante. Por el contrario, en el programa (d) no es posible detectar ningún valor para el acumulador cuanto se cumple que $n \geq 1$. Esto se debe que que el valor de $ac$ queda determinado por la relación no lineal $ac = k \times n$.

### 4.2.3. Ejemplo 3

Como tercer resultado obtenido se analizará el programa que se utiliza como ejemplo en [CH78].

La sintaxis concreta del programa es:

```
public void heapSortHalbwachs(int N, int[] T) {
    int L, R, I, J;

    int K;

    L = (N / 2) + 1;
2:  R = N;
    if (L >= 2) {
3:      L = L - 1;
        //K = T[L];
    } else {
        //K = T[R];
        //T[R] = 1;
5:      R = R - 1;
    }

    while (R >= 2) {
8:      I = L;
        J = 2 * I;
        while (J <= R) {
10:         if (J <= R - 1) {
                if (/*T[J] < T[J+1]*/new Object() == null) {
11:                 J = J + 1;
                }
            }
            if (/*K >= T[J]*/new Object() == null) {
                break;
            }
            //T[I] = T[J];
            I = J;
            J = 2 * J;
        }
15:     //T[I] = K;
        if (L >= 2) {
17:         L = L - 1;
            //K = T[L];
        } else {
            //K = T[R];
            //T[R] = T[1];
19:         R = R - 1;
        }
21:     //T[1] = k;
    }
}
```

Las sentencias que involucran referencias a *arrays* han sido comentadas al igual que se hizo en el trabajo original. Para simular condiciones no lineales se utilizó la expresión `new Object() == null`.

La versión *Jimple* del programa es:

```
public void heapSortHalbwachs(int, int[])
{
        input.Resultados this;
        int N, L, R, I, J, $i0, $i1;
        int[] T;
        java.lang.Object $r0;

        this := @this: input.Resultados;
        N := @parameter0: int;
        T := @parameter1: int[];
        if N < 2 goto label10;

        $i0 = N / 2;
        L = $i0 + 1;
 2:     R = N;
        if L < 2 goto label0;

 3:     L = L + -1;
        goto label9;

    label0:
 5:     R = R + -1;
        goto label9;

    label1:
 8:     I = L;
        J = 2 * I;
        goto label5;

    label2:
10:     $i1 = R - 1;
        if J > $i1 goto label3;

        $r0 = new java.lang.Object;
        specialinvoke $r0.<java.lang.Object: void <init>()>();
        if $r0 != null goto label3;

11:     J = J + 1;

    label3:
        $r0 = new java.lang.Object;
        specialinvoke $r0.<java.lang.Object: void <init>()>();
        if $r0 != null goto label4;

        goto label6;

    label4:
        I = J;
        J = J * 2;

    label5:
        if J <= R goto label2;

    label6:
```

```
15:     if L < 2 goto label7;

17:     L = L + -1;
        goto label9;

    label7:
19:     R = R + -1;

    label9:
        if R >= 2 goto label1;

    label10:
21:     return;
   }
```

El análisis se realizó, al igual que el análisis original, teniendo como precondición que $N \geq 2$. El resultado obtenido puede verse en la tabla de la figura 4.2.

| | |
|---|---|
| $2^{después}$ | $\{\ N = R \wedge \$i0 = L - 1 \wedge R \geq 2 \wedge R \geq 2L - 2 \wedge 2L \geq R + 1\ \}$ |
| $3^{antes}$ | $\{\ N = R \wedge \$i0 = L - 1 \wedge R \geq 2L - 2 \wedge 2L \geq R + 1 \wedge L \geq 2\ \}$ |
| $3^{después}$ | $\{\ N = R \wedge \$i0 = L \wedge R \geq 2L \wedge 2L \geq R - 1 \wedge L \geq 1\ \}$ |
| $5^{antes}$ | $\{$ FALSE $\}$ |
| $5^{después}$ | $\{$ FALSE $\}$ |
| $8^{antes}$ | $\{N = R \wedge \$i0 = L \wedge R \geq 2L \wedge 2L \geq R - 1 \wedge L \geq 1\} \cup$<br>$\{N \geq R \wedge \$i0 \geq L + 1 \wedge L \geq 1 \wedge R \geq 2\} \cup$<br>$\{L = 1 \wedge N \geq R + 1 \wedge \$i0 \geq 1 \wedge R \geq 2\}$ |
| $8^{después}$ | $\{\ L = I \wedge \$i0 \geq I \wedge N \geq R \wedge I \geq 1\ \}$ |
| $10^{antes}$ | $\{2L = J \wedge 2I = J \wedge J \geq 2 \wedge 2\$i0 \geq J \wedge N \geq R \wedge R \geq J\} \cup$<br>$\{R = \$i1 + 1 \wedge 2I = J \wedge L \geq 1 \wedge \$i0 \geq L \wedge N \geq \$i1 + 1 \wedge \$i1 \geq J - 1\}$ |
| $11^{antes}$ | $\{\ R = \$i1 + 1 \wedge 2I = J \wedge \$i0 \geq L \wedge L \geq 1 \wedge \$i1 \geq J \wedge N \geq \$i1 + 1\ \}$ |
| $11^{después}$ | $\{\ R = \$i1 + 1 \wedge 2I = J - 1 \wedge N \geq \$i1 + 1 \wedge \$i0 \geq L, L \geq 1 \wedge \$i1 \geq J - 1\ \}$ |
| $15^{antes}$ | $\{R = \$i1 + 1 \wedge N \geq \$i1 + 1 \wedge L \geq 1 \wedge \$i0 \geq L\} \cup$<br>$\{2L = J \wedge 2I = J \wedge J \geq 2 \wedge 2\$i0 \geq J \wedge N \geq R \wedge J \geq R + 1\}$ |
| $17^{antes}$ | $\{R = \$i1 + 1 \wedge N \geq \$i1 + 1 \wedge \$i0 \geq L \wedge L \geq 2\} \cup$<br>$\{2L = J \wedge 2I = J \wedge 2\$i0 \geq J \wedge N \geq R \wedge J \geq R + 1 \wedge J \geq 4\}$ |
| $17^{después}$ | $\{\ N \geq R \wedge \$i0 \geq L + 1 \wedge L \geq 1\ \}$ |
| $19^{antes}$ | $\{L = 1 \wedge R = \$i1 + 1 \wedge N \geq \$i1 + 1 \wedge \$i0 \geq 1\} \cup$<br>$\{L = 1 \wedge I = 1 \wedge J = 2 \wedge \$i0 \geq 1 \wedge N \geq R \wedge 1 \geq R\}$ |
| $19^{después}$ | $\{\ L = 1 \wedge N \geq R + 1 \wedge \$i0 \geq 1\ \}$ |
| $21^{antes}$ | $\{N \geq R \wedge \$i0 \geq L + 1 \wedge L \geq 1 \wedge 1 \geq R\} \cup$<br>$\{L = 1 \wedge N \geq R + 1 \wedge \$i0 \geq 1 \wedge 1 \geq R\}$ |

Figura 4.2: Restricciones descubiertas por la implementación.

Algunas observaciones pertinentes son las siguientes:

Al transformar el programa al lenguaje *Jimple* se agregan las variables $\$i0$ y $\$i1$. Para que las restricciones se ajusten al conjunto de variables del programa escrito en la sintaxis concreta, dichas variables se pueden eliminar (proyectar). El conjunto de restricciones que se obtiene al proyectar las variables temporales puede verse en la tabla de la figura 4.3.

La comparación entre el resultado que se obtiene sin proyectar las variables, y el que

| | |
|---|---|
| $2^{después}$ | $\{\ N \geq 2 \wedge N+1 \leq 2L \leq N+2 \wedge R = N\ \}$ |
| $3^{antes}$ | $\{\ L \geq 2 \wedge N+1 \leq 2L \leq N+2 \wedge R = N\ \}$ |
| $3^{después}$ | $\{\ L \geq 1 \wedge N-1 \leq 2L \leq N \wedge R = N\ \}$ |
| $5^{antes}$ | $\{$ FALSE $\}$ |
| $5^{después}$ | $\{$ FALSE $\}$ |
| $8^{antes}$ | $\{\ L \geq 1 \wedge R \geq 2 \wedge N \geq R\ \}$ |
| $8^{después}$ | $\{\ L = I \wedge I \geq 1 \wedge N \geq R\ \}$ |
| $10^{antes}$ | $\{\ 2I = J \wedge N \geq R \wedge R \geq J \wedge L \geq 1\ \}$ |
| $11^{antes}$ | $\{\ 2I = J \wedge R \geq J+1 \wedge N \geq R \wedge L \geq 1\ \}$ |
| $11^{después}$ | $\{\ 2I = J-1 \wedge L \geq 1 \wedge R \geq J \wedge N \geq R\ \}$ |
| $15^{antes}$ | $\{\ N \geq R \wedge L \geq 1\ \}$ |
| $17^{antes}$ | $\{\ N \geq R \wedge L \geq 2\ \}$ |
| $17^{después}$ | $\{\ N \geq R \wedge L \geq 1\ \}$ |
| $19^{antes}$ | $\{\ L = 1 \wedge N \geq R\ \}$ |
| $19^{después}$ | $\{\ L = 1 \wedge N >= R+1\ \}$ |
| $21^{antes}$ | $\{\ L \geq 1 \wedge 1 \geq R \wedge N \geq R\ \}$ |

Figura 4.3: Restricciones descubiertas por la implementación. Poyectando variables temporales

se obtiene proyectando las variables permite concluir que hay veces en que existe pérdida de información al proyectar variables. Por ejemplo, en el punto del programa $15^{antes}$ se observa que al proyectar las variables $\$i0$ y $\$i1$ se pierde toda la información que se había obtenido de las variables $I$ y $J$. Esto se debe a que, como se explica en la sección que describe el operador $\searrow$ (sección 2.1.1), al eliminar las dimensiones correspondientes a las variables $\$i0$ y $\$i1$ no quedan relaciones ni explícitas, ni implícitas, entre las variables $I$ y $J$.

Resultaría natural incluir en esta sección una comparación entre las restricciones obtenidas por nuestra implementación y las restricciones publicadas en [CH78] para el mismo programa. Sin embargo, lamentablemente, las restricciones publicadas en [CH78] tienen varios errores que hicieron que la comparación no tuviera sentido. Por ejemplo, la restricción para el punto del programa $2^{después}$ publicada en [CH78] es:

$$\{\ N \geq 2 \wedge N \leq 2L \leq N+1 \wedge R = N\ \}$$

Si se considera que $N = 4$, entonces en el punto $2^{después}$ se tiene que $L = 3$ y a partir de $2L \leq N+1$ se encuentra la contradicción $6 \leq 5$. Además, para el punto del programa $8^{antes}$ la restricción es:

$$\{\ R \geq 2 \wedge 2L \leq N+1 \wedge R+3 \leq N \wedge 2L+2R+1 \leq 3N \wedge L \geq 1 \wedge R \leq N\ \}$$

y si bien parecería más precisa que la obtenida por nuestra implementación, la relación $R+3 \leq N$ es falsa.

### 4.2.4. Ejemplo 4

Es este ejemplo se utilizará el lenguaje del capítulo 3 para realizar una comparación entre los dos tipos de análisis interprocedural presentados. Se analizarán dos programas, para poder obtener alguna conclusión al respecto

El primer programa es el que se muestra en la figura 4.4

```
    void inliningMasPrecisoSymbolic() {
1:      int a = imps(10, 100);
    }

    int imps(int p, int q) {
        int x = 1000;
        int ret = 3;

        while (p < q) {
            if (p < x) {
                while (x > 0) {
                    ret = 5;
                    x--;
                }
            } else {
                while (x > 0) {
                    ret = 7;
                    x--;
                }
            }
            p++;
        }
        return ret;
    }
```

Figura 4.4: Primer programa para comparar los dos tipos de análisis interprocedural.

La restricción obtenida utilizando *inlining* para el punto del programa $1^{después}$ es:

$\{\ a = 5\ \}$

La restricción obtenida utilizando análisis interprocedural simbólico para el mismo punto del programa es:

{ `TRUE` }

De esta forma se puede afirmar que el análisis interprocedural simbólico no es más preciso que el *inlining*, ya que se ha encontrado un contraejemplo para la afirmación.

La pérdida de precisión se debe a que al analizar el método *imps* utilizando análisis interprocedural simbólico se obtiene la siguiente post-condición:

$\{p = p_0 \wedge q = q_0 \wedge x = 1000 \wedge ret = 3 \wedge p_0 \geq q_0\} \cup$

$\{p = q_0 \wedge q = q_0 \wedge x \leq 0 \wedge q_0 \geq p_0 + 1\ \}$

lo que significa que al entrar al ciclo con la condición de corte `p < q` no se logra descubrir ningún tipo de información respecto a la variable *ret*, y esto hace que en el punto del programa $1^{después}$ tampoco se tenga ningún tipo de información al respecto de la variable $a$.

Habiendo analizado el programa de la figura 4.4 se podría conjeturar que el análisis interprocedural utilizando *inlining* es, al menos, tan preciso como el análisis interprocedural simbólico, y no se puede conjeturar lo contrario dado que se encontró un contraejemplo. Sin embargo, al analizar el programa de la figura 4.5 se obtienen las siguientes restricciones.

```
    void symbolicMasPrecisoInlining() {
1:      int a = smpi(100);
    }

    int smpi(int p)
    {
        int x = 0;
        while(x<=p)
        {
2:          x=x+1;
        }
        return x;
    }
```

Figura 4.5: Segundo programa para comparar los dos tipos de análisis interprocedural.

La restricción obtenida utilizando *inlining* para el punto del programa $1^{después}$ es:

$\{\ a \geq 101\ \}$

La restricción obtenida utilizando análisis interprocedural simbólico para el mismo punto del programa es:

$\{\ a = 101\ \}$

Esta diferencia se debe a que, al analizar simbólicamente, la restricción en el punto del programa $2^{después}$ contiene la relación $p_0 \geq x - 1$. Luego al intersecar la negación de la guarda, representada por la relación $p_0 \leq x - 1$, se llega a la relación $x = p_0 + 1$. Por otro, lado al analizar mediante *inlining*, se cuenta con los valores concretos de $x$ y $p$, y la restricción en el punto del programa $2^{después}$ pierde la relación entre ambas variables al realizar el *widening*.

Como puede observase, el resultado refuta la conjetura realizada anteriormente y la única conclusión con respecto a la comparación de la precisión de ambas técnicas de análisis interprocedural es que en algunos casos una resulta más precisa que la otra y en algunos otros, viceversa.

Por otro lado, este ejemplo también permite concluir que las restricciones descubiertas son un tanto más débiles que las restricciones de precisión absoluta. Esta imprecisión se debe a la utilización del operador de *widening* que aproxima la solución de las ecuaciones representada por el mínimo punto fijo (ver sección 3.4.1).

### 4.2.5. Ejemplo 5

En este ejemplo se muestra cómo ambas técnicas de análisis interprocedural se comportan correctamente cuando los métodos invocados retornan valores constantes que pueden ser expresados mediante una relación lineal o valores constantes que no pueden serlo.

En la figura 4.6 se presenta un programa en donde se realiza una invocación a un método que retorna un valor constante que puede ser representado mediante una relación lineal.

```
    void retValueCte() {
        int d = 10;
1:      d = retCte();
    }

    int retCte() {
       int a = 6;
       int b = 8;
       int c = 2 * a;
       int d = c + b;
       return d;
    }
```

Figura 4.6: El método invocado retorna una expresión constante lineal.

Las restricciones obtenidas utilizando ambas técnicas de análisis interprocedural son:

| | |
|---|---|
| $1^{antes}$ | $\{\ d = 10\ \}$ |
| $1^{después}$ | $\{\ d = 20\ \}$ |

En la figura 4.7 se presenta un programa en donde se realiza una invocación a un método que retorna un valor constante que no puede ser representado mediante una relación lineal.

```
    void retValueNoLineal() {
        int d = 10;
1:      d = retNoLineal();
    }

    int retNoLineal() {
       int a = 6;
       int b = (int)Math.sqrt(a);
       return b;
    }
```

Figura 4.7: El método invocado retorna una expresión constante no lineal.

Las restricciones obtenidas utilizando ambas técnicas de análisis interprocedural son:

| | |
|---|---|
| $1^{antes}$ | $\{\ d = 10\ \}$ |
| $1^{después}$ | $\{$ TRUE $\}$ |

### 4.2.6. Ejemplo 6

En este ejemplo se presentan programas en donde el análisis es capaz de descubrir la no terminación de los métodos invocados.

Las restricciones obtenidas para el programa de la figura 4.8 utilizando ambas técnicas de análisis interprocedural son:

```
    private void notermina1()
    {
1:      int a = nt1(5);
    }

    private int nt1(int a) {
        int i = 0;
        while (i < a) {
            a = i + 1;
        }
        return i;
    }
```

Figura 4.8: El método invocado no termina cuando el argumento es mayor a cero.

| | |
|---|---|
| $1^{antes}$ | { TRUE } |
| $1^{después}$ | { FALSE } |

Es interesante observar que la post-condición del método $nt1$ que se obtiene con la herramienta es:

$\{\ a = a_0 \wedge i = 0 \wedge retValue = 0 \wedge a_0 \leq 0\ \}$

Dicha post-condición refleja la necesidad de que el argumento sea menor o igual a cero para que el método termine.

```
    private void notermina2()
    {
1:      int x = nt2(5, 6);
2:      int y = nt2(5, 7);
 }

    private int nt2(int a, int b) {
        while (a < b) {
            if (a + 1 == b) {
                a++;
            } else {
                while (a + 1 > b) {
                    b--;
                }
            }
        }
        return a + b;
    }
```

Figura 4.9: El método invocado no termina para los valores de los argumentos presentes en la segunda invocación.

Las restricciones obtenidas para el programa de la figura 4.9 utilizando ambas técnicas de análisis interprocedural son:

| | |
|---|---|
| $1^{antes}$ | { TRUE } |
| $1^{después}$ | $\{\ x = 12\ \}$ |
| $2^{antes}$ | $\{\ x = 12\ \}$ |
| $2^{después}$ | { FALSE } |

En este último ejemplo se observa como un caso de no terminación que podría considerarse no trivial es descubierto por nuestra implementación aun realizando análisis interprocedural simbólico.

Es interesante observar que la post-condición del método $nt2$ que se obtiene con la herramienta es:

$\{a = a_0 \wedge b = b_0 \wedge retValue = a_0 + b_0 \wedge a_0 \geq b_0\} \cup$

$\{a = a_0 + 1 \wedge b = b_0 \wedge retValue = 2a_0 + 2 \wedge b_0 = a_0 + 1\}$

Dicha post-condición refleja la necesidad de que se cumpla que $\{a_0 \geq b_0\} \cup \{b_0 = a_0 + 1\}$ para que el método termine.

### 4.2.7. Ejemplo 7

Para este ejemplo se tomó como inspiración el programa de la figura 1 de [BGY05] adaptándolo para realizar únicamente operaciones con números enteros. El programa se presenta el la figura 4.10.

```
    void m0(int mc) {
1:    int a = m1(mc);
2:    int e = m2(2*mc, a);
    }

    Object[] m1(int k) {
3:      int b = 0;
4:      for(int i=1; i<=k; i++)  {
5:          b = m2(i, b);
        }
6:      return b;
    }
```

```
    Object[] m2(int n, int s) {
        int c, e;
7:      Object f = 0;
8:      for(int j=1; j<=n; j++) {
9:          if(j % 3 == 0) {
10:             c = j*2 + s;
            } else {
11:             c = s;
            }
12:         f = c;
        }
13:     e = 1;
14:     return f;
    }
```

Figura 4.10: Programa inspirado en el ejemplo de [BGY05].

A continuación se muestran los resultados obtenidos en los puntos del programa que ha sido considerados importantes. La letra "I" indica que el resultado ha sido obtenido mediante simulación de *inlining* y la letra "S" indica que se ha realizado análisis interprocedural simbólico. El signo "-" se utiliza en los puntos del programa donde no hay análisis interprocedural.

| | | |
|---|---|---|
| $1^{después}$ | I | $\{\ mc = mc_0\ \}$ |
| | S | $\{\ mc = mc_0 \wedge mc_0 \leq 0 \wedge a = 0\ \} \cup \{\ mc = mc_0 \wedge mc_0 \geq 1\ \}$ |
| $2^{después}$ | I | $\{mc = mc_0 \wedge 2mc \leq 0 \wedge e = 0\} \cup$<br>$\{mc = mc_0 \wedge 2mc \geq 1 \wedge e = a\} \cup$<br>$\{mc = mc_0 \wedge 2mc \geq 1 \wedge e = a + 4mc\}$ |
| | S | $\{mc = mc_0 \wedge mc_0 \leq 0 \wedge e = 0 \wedge a = 0\} \cup$<br>$\{mc = mc_0 \wedge mc_0 \geq 1 \wedge e = a\} \cup$<br>$\{mc = mc_0 \wedge mc_0 \geq 1 \wedge e = a + 4mc\}$ |
| $5^{antes}$ | I | $\{k = k_0 \wedge k_0 \geq 1 \wedge i = 1 \wedge b = 0\} \cup \{k = k_0 \wedge k_0 \geq i\}$ |
| | S | $\{k = k_0 \wedge k_0 \geq 1 \wedge i = 1 \wedge b = 0\} \cup \{k = k_0 \wedge i \geq 2 \wedge k_0 \geq i\}$ |
| $5^{después}$ | I | $\{k = k_0 \wedge k_0 \geq i\}$ |
| | S | $\{k = k_0 \wedge k_0 \geq i \wedge i \geq 1\}$ |
| $6^{antes}$ | I | $\{k = k_0 \wedge k_0 \leq 0 \wedge b = 0 \wedge i = 1\} \cup \{k = k_0 \wedge i = k_0 + 1\}$ |
| | S | $\{k = k_0 \wedge k_0 \leq 0 \wedge b = 0 \wedge i = 1\} \cup \{k = k_0 \wedge i = k_0 + 1 \wedge i \geq 2\}$ |
| $9^{antes}$ | - | $\{n = n_0 \wedge s = s_0 \wedge f = 0 \wedge j = 1 \wedge n_0 \geq 1\} \cup$<br>$\{n = n_0 \wedge s = s_0 \wedge f = c \wedge c = s \wedge n_0 \geq j \wedge j \geq 2 \wedge 3 \times tmp + 3 \geq j\} \cup$<br>$\{n = n_0 \wedge s = s_0 \wedge f = c \wedge c = 6 \times tmp + s \wedge j = 3 \times tmp + 1 \wedge$<br>$n_0 \geq 3 \times tmp + 1 \wedge 3 \times tmp \geq 1\}$ |
| $10^{antes}$ | - | $\{n = n_0 \wedge s = s_0 \wedge j = 3 \times tmp \wedge j \geq 1 \wedge n_0 \geq j\}$ |
| $11^{antes}$ | - | $\{n = n_0 \wedge s = s_0 \wedge n_0 \geq j \wedge j \geq 1 \wedge j \leq 3 \times tmp - 1\} \cup$<br>$\{n = n_0 \wedge s = s_0 \wedge n_0 \geq j \wedge j \geq 1 \wedge j \leq 3 \times tmp + 2 \wedge j \geq 3 \times tmp + 1\}$ |
| $12^{después}$ | - | $\{n = n_0 \wedge s = s_0 \wedge c = 2 \times j + s \wedge f = c \wedge j = 3 \times tmp \wedge n_0 \geq j \wedge j \geq 1\} \cup$<br>$\{n = n_0 \wedge s = s_0 \wedge c = s \wedge f = c \wedge j \geq 1 \wedge n_0 \geq j \wedge j \leq 3 \times tmp + 2\}$ |
| $13^{después}$ | - | $\{n = n_0 \wedge s = s_0 \wedge f = 0 \wedge j = 1 \wedge e = 1 \wedge n_0 \geq 0\} \cup$<br>$\{n = n_0 \wedge s = s_0 \wedge n = j - 1 \wedge c = s \wedge f = c \wedge e = 1 \wedge$<br>$j \leq 3 \times tmp + 3 \wedge j \geq 2\} \cup$<br>$\{n = n_0 \wedge s = s_0 \wedge n = j - 1 \wedge c = 6 \times tmp + s \wedge j = 3 \times tmp + 1 \wedge$<br>$f = c \wedge e = 1 \wedge 3 \times tmp \geq 1\}$ |

Se puede observar como realizando simulación de *inlining* no es posible obtener información útil para el punto del programa $1^{después}$. Por otro lado, realizando análisis interprocedural simbólico, al menos se logra obtener la relación $a = 0$ para el caso en el cual no se ingresa al ciclo del método $m1$. Esto se debe a que el operador de *widening* resulta clave en el análisis de los ciclos, y en este caso, no es lo suficientemente poderoso como para obtener información en presencia de ciclos anidados (notar que la invocación al método $m2$ en la sentencia "`5:`" genera un ciclo anidado).

En el punto del programa $2^{después}$, donde solamente hay un único ciclo, ambas técnicas de análisis interprocedural logran obtener las relaciones exactas entre las variables $e$ y $a$ del método $m0$.

La variable $tmp$ que aparece en las restricciones $9^{antes}$, $10^{antes}$, $12^{después}$ y $13^{después}$ es la variable temporal que se introduce para representar mediante una restricción lineal la relación que involucra el operador módulo ("$\%$"). Es interesante observar como en la restricción correspondiente al punto del programa $10^{después}$ se descubre la relación $j = 3 \times tmp$, lo que indica que $j \,\%\, 3 = 0$. De la misma forma, en la restricción correspondiente al punto del programa $11^{después}$ se utiliza la relación $j \leq 3 \times tmp + 2 \wedge j \geq 3 \times tmp + 1$ para indicar que se cumple la relación $j \,\%\, 3 \neq 0$.

# Capítulo 5

# Aproximación al análisis en programas con características de orientación a objetos

En este capítulo se presentan de forma exploratoria una serie de ideas que podrían permitir realizar el descubrimiento de relaciones lineales entre variables de programas escritos en lenguajes con características de orientación a objetos. Se utiliza la descripción "lenguajes con características de orientación a objetos" y no "lenguajes orientados a objetos" para evitar cualquier posible discusión que puede surgir por la utilización de tipos de datos básicos o primitivos. En este sentido, en el lenguaje que se presenta en este capítulo no toda entidad en el programa es un objeto, sino que se podrán utilizar variables primitivas de tipo entero de la misma forma que sucede, por ejemplo, en el lenguaje de programación Java.

El la sección 5.1 se extiende el lenguaje de los programas de números enteros para permitir la definición de clases, la instanciación de objetos y la manipulación de referencias a objetos. En la sección 5.2 se definirá una estructura con operaciones que permitirá abstraer el *heap* para reflejar la instanciación de objetos y la manipulación de referencias. En la sección 5.3 se hace una presentación sin demasiado rigor formal de un conjunto de reglas que permiten construir ecuaciones de *Data Flow* cuya solución constituiría el resultado del análisis. En la sección 5.4 se considerará un conjunto adicional de características del lenguaje que se consideran "avanzadas" donde, entre otras cosas, se tratará el análisis de *arrays* y el análisis en presencia de variables estáticas. En la sección 5.5 se estudia la interacción entre la parte del análisis relacionada a dominios de poliedros convexos y la relacionada a la abstracción del *heap*. En la sección 5.6 se discute una posible alternativa para lograr la correcta formalización de las ideas presentadas en las secciones anteriores. Finalmente, en las secciones 5.7 y 5.8 se presentan las características de la herramienta que implementa las ideas presentadas en este capítulo y algunos resultados obtenidos a partir de dicha herramienta.

## 5.1. El lenguaje de los programas con características de orientación a objetos

Los programas con características de orientación a objetos sobre los que se realizará el análisis son los escritos utilizando la siguiente sintaxis:

| | | |
|---|---|---|
| Programa | → | Clases |
| Clases | → | Clase |
| Clases | → | Clases Clase |
| Clase | → | `class` *nombreClase* `extends` *nombreClase* `{` Campos Métodos `}` |
| Clase | → | `class` *nombreClase* `{` Campos Métodos `}` |
| Campos | → | Declaración |
| Campos | → | Campos Declaración |
| Declaración | → | DeclaraciónEntera |
| Declaración | → | DeclaraciónReferencia |
| DeclaraciónEntera | → | `int` *nombreVariable* |
| DeclaraciónReferencia | → | *nombreClase nombreVariable* |
| Métodos | → | Método |
| Métodos | → | Métodos Método |
| Método | → | *nombreMétodo* `()` `{` SentenciaEtiquetada `}` |
| Método | → | *nombreMétodo* `(` Parámetros `)` `{` SentenciaEtiquetada `}` |
| Parámetros | → | Declaración |
| Parámetros | → | Parámetros, Declaración |
| SentenciaEtiquetada | → | (Etiqueta, Sentencia) |
| SentenciaEtiquetada | → | (Etiqueta, Sentencia); SentenciaEtiquetada |
| Etiqueta | → | $n$ con $n \in \mathbb{N}$ |
| Sentencia | → | `if (` ExpresiónBooleana `)` `goto` Etiqueta |
| Sentencia | → | `goto` Etiqueta |
| Sentencia | → | `return` |
| Sentencia | → | `return` Operando |
| Sentencia | → | AsignaciónEntera |
| Sentencia | → | AsignaciónReferencia |
| Sentencia | → | Invocación |
| ExpresiónBooleana | → | *nombreVariable* OperadorBooleano OperandoEntero |
| ExpresiónBooleana | → | OperandoEntero OperadorBooleano *nombreVariable* |

ExpresiónBooleana → ExpresiónBooleanaNoLineal

OperadorBooleano → `==`
OperadorBooleano → `!=`
OperadorBooleano → `>=`
OperadorBooleano → `>`
OperadorBooleano → `<=`
OperadorBooleano → `<`

AsignaciónEntera → *nombreVariable* `=` ExpresiónZ
AsignaciónEntera → *nombreVariable*$_1$`.`*nombreVariable*$_2$ `=` *nombreVariable*$_3$

ExpresiónZ → $z$ con $z \in \mathbf{Z}$
ExpresiónZ → *nombreVariable*
ExpresiónZ → *nombreVariable*$_1$`.`*nombreVariable*$_2$
ExpresiónZ → `-` *nombreVariable*
ExpresiónZ → *nombreVariable* OperadorZ OperandoEntero
ExpresiónZ → OperandoEntero OperadorZ *nombreVariable*
ExpresiónZ → *nombreVariable* `*` $z$
ExpresiónZ → $z$ `*` *nombreVariable*
ExpresiónZ → ExpresiónZNoLineal
ExpresiónZ → Invocación

OperadorZ → `+`
OperadorZ → `-`

AsignaciónReferencia → *nombreVariable* `=` ExpresiónReferencia
AsignaciónReferencia → *nombreVariable*$_1$`.`*nombreVariable*$_2$ `=` *nombreVariable*$_3$

ExpresiónReferencia → *nombreVariable*
ExpresiónReferencia → *nombreVariable*$_1$`.`*nombreVariable*$_2$
ExpresiónReferencia → `new` *nombreClase*
ExpresiónReferencia → `null`
ExpresiónReferencia → Invocación

Invocación → *nombreVariable*`.`*nombreMétodo*`()`
Invocación → *nombreVariable*`.`*nombreMétodo*`(`Argumentos`)`

Argumentos → Operando
Argumentos → Argumentos, Operando

Operando → OperandoEntero
Operando → OperandoReferencia

OperandoEntero → *nombreVariable*
OperandoEntero → $z$

OperandoReferencia → *nombreVariable*
OperandoReferencia → `null`

Al igual que el de la sección 3.1 este lenguaje también está inspirado en *Jimple* [VRHS+99], que es una representación intermedia del lenguaje Java. Además, nuevamente se asumirá que los programas a analizar cumplen con todas las restricciones semánticas necesarias, ya que se asumen obtenidos a partir de programas escritos en una sintaxis concreta que superan exitosamente el chequeo sintáctico y posteriormente los chequeos semánticos correspondientes.

Las categorías gramaticales ExpresiónZNoLineal y ExpresiónBooleanaNoLineal no son expandidas debido a que, a los efectos del análisis, todas sus posibles expresiones tienen asociada la misma semántica.

Una aclaración pertinente es que se asume que la sentencia que crea nuevas instancias de objetos en memoria (`new`), solamente reserva el espacio necesario de memoria dinámica, sin realizar ninguna inicialización. La inicialización que puede encontarse, por ejemplo, en los constructores del lenguaje Java se representa mediante invocación estándar de algún método distinguido luego de la instanciación.

## 5.2. Abstracción del *heap*

Al realizar el descubrimiento de restricciones lineales entre variables en un lenguaje de las características del presentado en la sección 5.1, existe un conjunto de cuestiones a considerar que no estaban presentes en el análisis de programas de números enteros.

```
    class A { int f; }

    class B {
        m1(int p1) {
1:          A a = new A();
2:          A b = null;
3:          A c = null;

4:          a.f = 0;
            if (p1 > 0) {
5:              b = a;
6:              b.f = 1;
            } else {
7:              c = a;
8:              c.f = 2;
            }
9:          a.f = 3;
        }
    }
```

Figura 5.1: Programa sencillo que realiza instanciación de objetos utilizando una hipotética sintaxis concreta

Si se considera, por ejemplo, el programa de la figura 5.1 escrito en una hipotética sintaxis concreta se puede observar lo siguiente.

La restricción después de la sentencia "`4:`" es $\{\ a.f = 0\ \}$ indicando que el campo $f$ del objeto $a$ tiene el valor 0. Por otro lado la restricción después de la sentencia "`5:`" es

$\{\ a.f = 0 \wedge b.f = 0 \wedge p1 > 10\ \}$, ya que el objeto $b$ pasa a ser el mismo que el objeto $a$. Después de la sentencia "`6:`" la restricción es $\{\ a.f = 1 \wedge b.f = 1 \wedge p1 > 10\ \}$, ya que $b$ y $a$ son el mismo objeto. Esta situación se conoce como *aliasing* y básicamente consiste en la posibilidad de utilizar diferentes nombres para referirse a la misma posición de memoria. Una situación análoga ocurre con las sentencias "`7:`" y "`8:`". Sin embargo, podría ser un poco más complicado de definir cual es la restricción antes y después de la sentencia "`9:`". Una posibilidad es $\{\ a.f = 1 \wedge b.f = 1 \wedge p1 > 10\ \} \cup \{\ a.f = 2 \wedge c.f = 2 \wedge p1 \leq 10\ \}$ para antes de la sentencia y $\{\ a.f = 3 \wedge b.f = 3 \wedge p1 > 10\ \} \cup \{\ a.f = 3 \wedge c.f = 3 \wedge p1 \leq 10\ \}$ para después de la sentencia. La situación podría adquirir más complejidad si la sentencia "`9:`" es reemplazada por `c.f = 3`, donde si se cumple que $p1 > 10$ se estaría "dereferenciando" un puntero nulo.

Como puede verse el análisis en presencia de instanciación de objetos y *aliasing* resulta mucho más complejo que el análisis de programas de números enteros ya que, entre otras cosas, una única sentencia podría estar modificando varias variables de la restricción. Además, determinar cuáles son las variables que se están modificando no es una tarea sencilla, ya que se debe tener en cuenta todas las posibles referencias que apuntan a al mismo objeto.

Para facilitar la tarea de realizar el análisis en este contexto resulta útil separar la información y tener, por un lado, las restricciones lineales como se las viene considerando hasta este momento, y por otro lado una estructura que abstraiga toda la complejidad de la creación de objetos en memoria y la manipulación de referencias. Esta idea está inspirada en la que se presenta en [CL05a], donde se utiliza una estructura independiente para representar y manipular las llamadas *alien expressions*, que básicamente consisten en las expresiones que están fuera del alcance del dominio del análisis.

En nuestro trabajo la definición y manipulación de objetos se considera fuera del dominio de las restricciones lineales entre variables enteras, y se abstrae con una construcción adicional que se llamara *AHeap* (por abstracción del *heap*) y se define a continuación.

$$AHeap = (HeapMap, Nulls, WeakUpdates)$$

$$HeapMap = MemoryLocation \rightarrow \wp(Referencia)$$

$$Nulls \in \wp(Referencia)$$

$$WeakUpdates \in \wp(MemoryLocation)$$

$MemoryLocation$ y $Referencia$ en este punto deben ser considerados cadenas de caracteres que se utilizan para denotar posiciones de memoria y referencias (o punteros) a dichas posiciones, respectivamente. Más adelante se verá que resulta conveniente que estos dos elementos posean una estructura más compleja, pero para facilitar la comprensión es suficiente en esta etapa considerarlos simplemente nombres.

La abstracción del *heap* está compuesta de tres elementos: $HeapMap$, $Nulls$ y $WeakUpdates$.

$HeapMap$ es una función que a cada $MemoryLocation$ le hace corresponder un conjunto de elementos $Referencia$. $MemoryLocation$ es la abstracción de un objeto instanciado en memoria, y el correspondiente conjunto de elementos $Referencia$ representa precisamente las referencias que apuntan (o pueden apuntar) a dicho objeto. La expresión $heapMap_i(ml_j)$ representa el conjunto de referencias que apuntan a la posición de memoria $ml_j$ según el mapeo $heapMap_i$, con la precondición de que $ml_j \in Dom(heapMap_i)$ (o sea la posición de memoria debe pertenecer al dominio de la función).

$Nulls$ es el conjunto de referencias que son nulas (o pueden serlo). Es necesario decir que una referencia **puede o no** apuntar a un objeto, o **puede o no** ser nula, porque el análisis se comporta de forma conservadora y muchas veces no es posible realizar la afirmación correspondiente. Por ejemplo, en el programa de la figura 5.1 antes de la sentencia "`9:`", la referencia $c$ puede o no ser nula, en cambio es posible afirmar que la referencia $a$ no es nula. En consecuencia la referencia $c$ pertenecerá al conjunto $Nulls$, pero la $a$ no.

Finalmente la utilidad del conjunto $WeakUpdates$ será explicada en detalle más adelante, pero es posible mencionar que está relacionada con un concepto muy utilizado al realizar análisis de programas que es el de *strong updates* y *weak updates*. La idea es que en presencia de un *strong update* los valores viejos de la entidad que se actualiza dejan de existir y son reemplazados por los nuevos. En cambio, en un *weak update* los valores viejos de la entidad que se actualiza "conviven" de alguna forma con los nuevos valores.

A continuación se describen utilizando pseudo-código una serie de operaciones sobre $AHeap$ que serán utilizadas para representar la manipulación de objetos y referencias durante el análisis.

$getPointedMemLocs(AHeap, Referencia) : \wp(MemoryLocation)$

ENTRADA: $aHeap_1 = (heapMap_1, nulls_1, weakUpdates_1)$,
$refenrencia_1$

SALIDA: $ret_1 \in \wp(MemoryLocation)$

ALGORITMO: $ret_1 = \emptyset$
para cada $(memoryLocation_j \rightarrow R_j) \in heapMap_1$
si $referencia_1 \in R_j$ entonces $ret_1 = ret_1 \cup \{memoryLocation_j\}$;

Este método recibe una abstracción del *heap* y una referencia, y retorna el conjunto de posiciones de memoria que son, o pueden ser, apuntadas por la referencia indicada.

Otra forma de representar en conjunto "salida" del algoritmo anterior es mediante la expresión:

$\{ml | referencia_1 \in heapMap_1(ml)\}$

$eliminateRef(AHeap, Referencia) : AHeap$

ENTRADA: $aHeap_1 = (heapMap_1, nulls_1, weakUpdates_1)$,
$refenrencia_1$

SALIDA: $aHeap_{ret} = (heapMap_{ret}, nulls_{ret}, weakUpdates_{ret})$

ALGORITMO: $aHeap_{ret} = aHeap_1$
para cada $(MemoryLocation_j \rightarrow R_j) \in heapMap_{ret}$
$R_{tmp} = R_j \backslash \{refenrencia_1\}$
$heapMap_{ret} = (heapMap_{ret} \backslash MemoryLocation_j \rightarrow R_j) \cup$
$\{MemoryLocation_j \rightarrow R_{tmp}\}$
$nulls_{ret} = nulls_{ret} \backslash \{refenrencia_1\}$

Este método recibe una abstracción del *heap* y una referencia, y elimina todo rastro de la referencia del *heap*. Cabe aclarar que es posible que el conjunto de referencias que apunte a alguna posición de memoria quede vacío. Esto es análogo a la situación que se presenta cuando queda memoria en desuso sin ser liberada.

La abstracción del *heap* que representa la "salida" del algoritmo también puede ser representada por la expresión:

$\{(ml, r \backslash referencia_1) | (ml, r) \in heapMap_1, nulls_1 \backslash referencia_1, weakUpdates_1\}$

$addNewEntry(AHeap, MemoryLocation, Referencia) : AHeap$

ENTRADA: $aHeap_1 = (heapMap_1, nulls_1, weakUpdates_1)$,
$ml_1$,
$ref_1$,

SALIDA: $aHeap_{ret} = (heapMap_{ret}, nulls_{ret}, weakUpdates_{ret})$

ALGORITMO: $aHeap_{ret} = aHeap_1$
si $(ml_1 \in Dom(heapMap_1))$
$weakUpdates_{ret} = weakUpdates_{ret} \cup \{ml_i\}$;
$tmp = heapMap_{ret}(ml_1) \cup \{ref_1\}$;
$heapMap_{ret} = heapMap_{ret} \backslash \{(ml_1 \rightarrow heapMap_{ret}(ml_1))\} \cup$
$\{\ (ml_1 \rightarrow tmp)\}$;
sino
$aHeap_{ret} = aHeap_{ret} \cup \{(ml_1 \rightarrow \{ref_1\})\}$

Esta operación es utilizada para agregar una nueva $MemoryLocation$ al $AHeap$ que será apuntada por la $Referencia$ indicada. En caso de que la posición de memoria ya exista en el mapeo será agregada a la lista de *weak updates*. Esto se debe a que una misma posición de memoria de *Aheap* se utiliza para representar más de un objeto que potencialmente se instanciará en *runtime*.

$addAlias(AHeap, Referencia, Referencia, boolean) : AHeap$

ENTRADA: $aHeap_1 = (heapMap_1, nulls_1, weakUpdates_1)$,
$ref_{target}$,
$ref_{source}$,
$weak_1$

SALIDA: $aHeap_{ret} = (heapMap_{ret}, nulls_{ret}, weakUpdates_{ret})$

ALGORITMO: $aHeap_{ret} = aHeap_1$
si $(weak_1)$ entonces
si $(getPointedMemLocs(aHeap_1, ref_{target}) = \emptyset)$ entonces
$nulls_{ret} = nulls_{ret} \cup \{ref_{target}\}$;
sino
$aHeap_{ret} = eliminateRef(aHeap_{ret}, ref_{target})$;
para cada $(MemoryLocation_j \rightarrow R_j) \in heapMap_1$
si $ref_{source} \in R_j$ entonces
$heapMap_{ret} = heapMap_{ret} \backslash \{(MemoryLocation_j \rightarrow R_k)\} \cup$
$\{ (MemoryLocation_j \rightarrow (R_k \cup \{ref_{target}\}))\}$;
si $(ref_{source} \in nulls_1)$
$nulls_{ret} = nulls_{ret} \cup \{ref_{target}\}$;

Esta operación es utilizada para hacer que una referencia (destino) sea un alias de otra (fuente). El último argumento de tipo booleano indica el tipo de actualización que debe hacerse, o sea, *weak update* o *strong update*. En caso de *weak update* se debe comprobar si la referencia apunta a otra posición de memoria. En caso negativo es necesario agregar la referencia al conjunto de referencias nulas. Esto se debe a que como se está en presencia de un *weak update* no debe perderse la información previa a la incorporación de la nueva entrada, y si no apunta a ninguna posición de memoria, se asume que la referencia era nula. En caso de un *strong update* simplemente se elimina todo rastro de la referencia utilizando la operación $eliminateRef$. Luego, se agrega la referencia destino a cada conjunto de referencias a los que pertenezca la referencia fuente. Un detalle que cabe mencionar es que es necesario realizar la comprobación de pertenencia o no de la referencia fuente a un conjunto de referencias teniendo en cuenta el mapeo original ($heapMap_1$). Si dicha comprobación se realizase teniendo en cuenta $heapMap_{ret}$ se puede presentar un error si se realiza un *strong update* y la referencia fuente y destino están relacionadas[1].

[1] dos referencias están relacionadas por ejemplo en la sentencia `nodo = nodo.next`, donde se ve que una referencia apunta a un campo de la otra

$addNullAlias(AHeap, Referencia) : AHeap$

ENTRADA: $aHeap_1 = (heapMap_1, nulls_1, weakUpdates_1)$,
$ref_{target}$,

SALIDA: $aHeap_{ret} = (heapMap_{ret}, nulls_{ret}, weakUpdates_{ret})$

ALGORITMO: $aHeap_{ret} = aHeap_1$
$nulls_{ret} = nulls_{ret} \cup \{ref_{target}\}$;

Esta operación es utilizada para agregar una referencia al conjunto de referencias nulas.

$union(AHeap, AHeap) : AHeap$

ENTRADA: $aHeap_1 = (heapMap_1, nulls_1, weakUpdates_1)$,
$aHeap_2 = (heapMap_2, nulls_2, weakUpdates_2)$

SALIDA: $aHeap_{ret} = (heapMap_{ret}, nulls_{ret}, weakUpdates_{ret})$

ALGORITMO: $heapMap_{ret} = heapMap_1$;
para cada $(ml_j \rightarrow R_j) \in heapMap_2$
si $ml_j \in Dom(heapMap_{ret})$ entonces
$tmp = heapMap_{ret}(ml_j) \cup R_j$;
$heapMap_{ret} = heapMap_{ret} \backslash \{(ml_j \rightarrow heapMap_{ret}(ml_j))\} \cup$
$\{ (ml_j \rightarrow tmp)\}$;
sino
$heapMap_{ret} = heapMap_{ret} \cup (ml_j \rightarrow R_j)$;
$weakUpdates_{ret} = nulls_1 \cup nulls_2$;
$weakUpdates_{ret} = weakUpdates_1 \cup weakUpdates_2$;

Esta operación es utilizada "unir" la información de dos abstracciones del *heap*. Desde el punto de vista lógico se comporta como la disyunción de ambas estructuras. Por ejemplo, si en una abstracción del *heap* la variable $a$ apunta a la posición de memoria $m1$, y en la otra abstracción del *heap* la variable $a$ apunta a la posición de memoria $m2$, en la unión de las estructuras la variable $a$ podrá apuntar a $m1$ o $m2$.

### 5.2.1. *Memory location*

En las operaciones presentadas anteriormente $MemoryLocation$ era un elemento que se presentaba como atómico. Sin embargo, para el resto del análisis es necesario definir $MemoryLocation$ de la siguiente manera:

$$MemoryLocation = (Etq, nombreClase, variante, serie)$$

La primera componente representa la etiqueta de la sentencia que está asociada a la posición de memoria. Se asume que existe una etiqueta "null" que se utiliza cuando la

posición de memoria no está relacionada a ninguna sentencia en particular del método que se analiza.

La segunda componente es el nombre de la clase asociada a la posición de memoria. A partir de este valor, se puede decir que las posiciones de memoria son "tipadas" y tienen asociada una clase particular.

La tercera componente es utilizada para representar el contexto en el cual existe la posición de memoria. Los contextos existentes son los siguientes:

- *Creation site*: utilizado para posiciones de memoria que son creadas utilizando la sentencia `new` en el método que se analiza.
- *Symbolic argument*: utilizado para posiciones de memoria que representan simbólicamente los objetos que el método puede recibir como argumentos.
- *Discovered argument field*: utilizado para posiciones de memoria que representan los objetos que son campos de un argumento simbólico.
- *This reference*: utilizado para representar simbólicamente la posición de memoria que ocuparía el objeto sobre el cual se realiza la hipotética invocación del método que se analiza.
- *Discovered this field*: utilizado para las posiciones de memoria que ocupan los objetos que son campos del objeto sobre el que se ejecuta el análisis.

Finalmente, la serie es un número entero utilizado para distinguir entre dos posiciones de memoria diferentes que tienen asociada la misma etiqueta.

A modo de ejemplo, la figura 5.2 muestra un programa, en una hipotética sintaxis concreta, donde, si se considera que el método que se analiza es $m1$, se presentan los diferentes contextos en los que se encuentran las variantes de posiciones de memoria descriptas.

```
    class B {
        int f;
        B   next;
    }

    class A {

        A f1;

        m1(B argB, B argB2) {
1:          A a = new A();
2:          int c1 = argB.f;
3:          int c2 = argB.next.f;
4:          A a2 = this;
5:          A a3 = this.f1;
        }
    }
```

Figura 5.2: Contextos de posiciones de memoria.

En la sentencia "`1:`" se presenta el caso en que la abstracción del *heap* debe modelar un nuevo objeto creado en el *scope* del método que se analiza (*Creation site*). En la sentencia "`2:`" se muestra la necesidad de representar en la abstracción del *heap* una posición de memoria "referenciada" por el parámetro $argB$ (*Symbolic argument*). En la sentencia "`3:`" ocurre lo mismo pero para campos de objetos que son argumentos simbólicos del método que se analiza (*Discovered argument field*). En la sentencia "`4:`" se muestra la necesidad de modelar posiciones de memoria de la variante *This reference* para que sean apuntadas, en el caso del ejemplo, por la variable $a2$. Finalmente la sentencia "`5:`" muestra el contexto *Discovered this field*.

La necesidad de distinguir entre estas cinco variantes de posiciones de memoria será aclarada más adelante cuando se desarrollen las reglas del análisis.

Para denotar las posiciones de memoria se utiliza la siguiente tabla de abreviaturas:

| | |
|---|---|
| *Creation site* | CS |
| *Symbolic argument* | SA |
| *Discovered argument field* | DA |
| *This reference* | TR |
| *Discovered this field* | DT |

La notación para representar una posición de memoria es:

$$variante_{Etq}^{nombreClase}$$

Cuando sea necesario se representará la diferencia en el número de serie de la posición de memoria mediante la utilización de símbolos prima (').

Por ejemplo, la posición de memoria que se crea en la sentencia "`1:`" del programa de la figura 5.2 es denotada:

$$CS_1^A$$

la posición de memoria relacionada a la sentencia "`2:`" es:

$$CS_{null}^B$$

donde se ve que las posiciones de memoria de los argumentos simbólicos no están relacionadas a ninguna sentencia del programa en particular.

la posición de memoria correspondiente al segundo argumento ($argB2$) es:

$$CS_{null'}^B$$

## 5.3. Ecuaciones de *Data Flow*

Al igual que el análisis interprocedural de programas de números enteros (sección 3.3), en este caso también se considera como unidad del análisis al método. Sin embargo,

a diferencia del análisis de programas enteros, las incógnitas de las ecuaciones no serán dominios de poliedros convexos cerrados, sino que será el par compuesto por un dominio y una abstracción del *heap*.

$$(Dominio, AHeap)$$

Para interpretar la información del análisis se deben tener en cuenta la información provista por ambos componentes del par. Por ejemplo, considerando el programa de la figura 5.1, la restricción después de la sentencia "`9:`" es:

$$(\{p1 = p1_0 \wedge cs_1^A.f = 3\}, (\{cs_1^A \rightarrow \{a, b, c\}\}, \{b, c\}, \emptyset))$$

En la primera componente de la restricción se ve que existe un *binding* entre el parámetro $p1$ y la meta-variable correspondiente, y que el objeto de clase $A$ instanciado en la sentencia "`1:`" tiene en el campo $f$ el valor 3. Por otro lado, la abstracción del *heap* dice que la variable $a$ apunta indefectiblemente al objeto mencionado, que $b$ puede apuntar a dicho objeto, o ser nula y para $c$ ocurre lo mismo que para $b$. A partir de esta restricción se deduce que $a.f = 3$, que $b.f = 3$ o $b = null$ y que $c.f = 3$ o $c = null$. Lamentablemente, de esta manera no es posible representar el hecho de que $b = null$ o $c = null$, donde la disyunción es exclusiva, o sea ambos no pueden ser nulos a la vez. En la sección 5.4.3 se presenta una mejora a este sistema de representación de restricciones, que permite mayor precisión en el resultado.

### 5.3.1. Resolución de referencias

La forma de representar la información utilizando el par $(Dominio, AHeap)$, hace que al momento de operar con una variable del programa sea necesario resolver las referencias, para saber qué posiciones de memoria están involucradas en la sentencia. Si se considera, por ejemplo, el programa de la figura 5.3 se ve que en la sentencia "`8:`" la variable $c$ puede estar apuntando a dos posiciones de memoria diferentes.

```
    class A { int f; }

    class B {
        m1(int p1) {
1:          A a = new A();
2:          A b = new A();
3:          A c = null;
4:          a.f = 0
5:          b.f = 1
            if (p1 > 0) {
6:              c = a;
            } else {
7:              c = b;
            }
8:          c.f = 3;
        }
    }
```

Figura 5.3: Ejemplo para resolución de referencias.

La restricción antes de la sentencia "`8:`" es:

$$(\{p1 = p1_0 \wedge cs_1^A.f = 0 \wedge cs_2^A.f = 1\}, (\{cs_1^A \rightarrow \{a, c\}, cs_2^A \rightarrow \{b, c\}\}, \emptyset, \emptyset))$$

Cuando se quieran representar los efectos de la ejecución de la sentencia "`8:`" se hace necesario resolver la referencia $c$ para saber a qué posiciones de memoria es posible que modifique dicha sentencia. Para esto se puede utilizar la operación $getPointedMemLocs(AHeap, Referencia)$, que en este caso arrojará como resultado el conjunto $\{cs_1^A, cs_2^A\}$. Como el conjunto tiene dos elementos la sentencia podrá modificar dos variables del análisis, y la restricción resultante es:

$$(\{p1 = p1_0 \wedge cs_1^A.f = 3 \wedge cs_2^A.f = 1\} \cup \{p1 = p1_0 \wedge cs_1^A.f = 0 \wedge cs_2^A.f = 3\},$$
$$(\{cs_1^A \rightarrow \{a, c\}, cs_2^A \rightarrow \{b, c\}\}, \emptyset, \emptyset))$$

Se llamará resolución de referencias al proceso que transforma la expresión $c.f$ en el conjunto de variables $\{cs_1^A.f, cs_2^A.f\}$ teniendo en cuenta la información brindada por una abstracción del *heap*.

Cabe aclarar que la resolución de referencias permite detectar estáticamente referencias a punteros nulos que en tiempo de ejecución producirán la liberación de una excepción o cualquier otra manifestación de anomalías. Esto da lugar a una extensión de la precisión del análisis permitiendo detectar errores en la programación.

### 5.3.2. Reglas

Nuevamente, las reglas para construir las ecuaciones de *Data Flow* pueden dividirse en dos grupos. El primer grupo es el que relaciona las incógnitas que representan restricciones antes de ejecutar una sentencia, con las restricciones después de ejecutar cada una de las sentencias que la anteceden en el flujo del programa. El segundo grupo de reglas relaciona las restricciones después de ejecutar una sentencia con la restricción existente antes de ejecutarla.

En todas las reglas se debe considerar que las incógnitas están representadas por el par $(Dominio, AHeap)$. Para poder referirse a alguna componente particular del par se utilizarán los siguientes proyectores:

$$D((dominio_i, aHeap_i)) = dominio_i$$

$$H((dominio_i, aHeap_i)) = aHeap_i$$

#### Primer conjunto de reglas

El primer grupo de reglas es el siguiente:

$$R_1^{oo}.\ \ Etq_i^{antes} = (D_{bindMetaArg}, H_{inicial}) \cup \bigcup_{(j,i)\in G_{ejes}^M} (procCond(D(Etq_j)), H(Etq_j))$$
$$\text{si } Etq_i = Prim(M)$$

$$R_2^{oo}.\ \ Etq_i^{antes} = \bigcup_{(j,i)\in G_{ejes}^M} (procCond(D(Etq_j)), H(Etq_j)) \text{ si } Etq_i \neq Prim(M)$$

donde $D_{bindMetaArg}$ es análogo al que se presenta en la sección 3.3.2. O sea, es el dominio que realiza el *binding* de cada parámetro formal de tipo `int` con un meta-argumento.

$H_{inicial}$ es una instancia de $AHeap$ que representa simbólicamente, por un lado, los objetos que son pasados como argumentos en una hipotética invocación y las referencias entre ellos, y por otro lado, la instancia del objeto sobre el cual se realiza el análisis.

Para representar simbólicamente cada argumento del tipo referencia, se utiliza una posición de memoria de la variante "SA" para cada argumento, y se la hace apuntar por el correspondiente parámetro formal. En caso de haber más de un argumento del mismo tipo deberán poseer número de serie diferentes. Por otro lado, se debe agregar cada parámetro formal a la lista de referencias nulas, indicando que existe la posibilidad de que la invocación se realice utilizando un puntero nulo. Para representar la instancia sobre la cual se realiza el análisis se utiliza una posición de memoria de la variante "TR" apuntada por la referencia distinguida *this*.

Sin embargo, esto sólo no es suficiente. Si se considera, por ejemplo un programa compuesto por las clases:

```
class A { ... }

class B extends A{

    m1(B arg1, A arg2) {
        ...
    }

}
```

y se analiza el método $m1$, sería un error considerar como $H_{inicial}$ a

$$(\{sa_{null}^B \rightarrow \{arg1\}, sa_{null}^A \rightarrow \{arg2\}, tr_{null}^B \rightarrow \{this\}\}, \{arg1, agr2\}, \emptyset)))$$

El error se debe a que la abstracción del *heap* no contempla la posibilidad de que la invocación se realice utilizando como argumentos la misma instancia de la clase B:

```
...
B argumento = new B;
argumento.m1(argumento, argumento)
...
```

Para solucionar este problema es necesario definir la relación:

$$esAsignableDesde : nombreClase \times nombreClase \rightarrow \{true, false\}$$

Esta relación es válida cuando la primera y la segunda clase son la misma, o cuando la segunda es una subclase de la primera. Donde una clase es subclase de otra de acuerdo con la relación de herencia tradicional en orientación a objetos. Dicha relación de herencia se genera en el lenguaje de este trabajo utilizando la definición de clases que contiene la palabra reservada `extends`.

Una vez definida esta relación se debe utilizar para hacer que cada parámetro formal y la variable distinguida *this* apunten, además de a su correspondiente posición de memoria "SA" y "TR", a cada una de las restantes posiciones de memoria con las que la relación *esAsignableDesde* sea verdadera.

En definitiva, la versión correcta de $H_{inicial}$ para el método $m1$ es:

$$(\{sa^B_{null} \rightarrow \{arg1, arg2, this\}, sa^A_{null} \rightarrow \{arg2\}, tr^B_{null} \rightarrow \{arg1, arg2, this\}\}, \{arg1, agr2\}, \emptyset)))$$

donde se ve que, como la clase $A$ es asignable desde $B$, y obviamente la clase $B$ es asignable desde $B$, el parámetro formal $arg2$ también debe apuntar a $sa^B_{null}$ y $tr^B_{null}$ para representar que existe la posibilidad de que ambas referencias sean en realidad al mismo objeto de clase $B$. Una situación análoga se presenta con la variable distinguida *this*.

Una aclaración con respecto a este primer conjunto de reglas es que las reglas utilizan el operador disyunción ("$\cup$"), pero no en dominios, sino en pares $(Dominio, AHeap)$. Este operador se define de la siguiente manera:

$$union((Dominio, AHeap), (Dominio, AHeap)) : (Dominio, AHeap)$$

ENTRADA: $(dominio_1, aHeap_1)$,
$(dominio_2, aHeap_2)$

SALIDA: $(dominio_{ret}, aHeap_{ret})$

ALGORITMO: $dominio_{ret} = dominio_1 \cup dominio_2$;
si $dominio_1 = D_{vacio} \wedge dominio_2 = D_{vacio}$ entonces
$aHeap_{ret} = (\emptyset, \emptyset, \emptyset)$;
sino si $dominio_1 = D_{vacio}$ entonces
$aHeap_{ret} = aHeap_2$;
sino si $dominio_2 = D_{vacio}$ entonces
$aHeap_{ret} = aHeap_1$;
sino
$aHeap_{ret} = union(aHeap_1, aHeap_2)$;

Como se ve en el algoritmo, la unión de los dominios se realiza de forma estándar, pero para unir las abstracciones del *heap* es necesario considerar que los dominios de los pares pueden ser vacíos y en ese caso, la abstracción del *heap* correspondiente no debe

aportar información. Este operador contribuye a lo que se llama *feedback* entre el dominio y la abstracción del *heap*, y sus efectos se estudian en la sección 5.5.

Cabe mencionar que a partir de la regla:

$$Etq_i^{antes} = \bigcup_{(j,i)\in G^M_{ejes}} (procCond(D(Etq_j)), H(Etq_j)) \text{ si } Etq_i \neq Prim(M)$$

puede decirse que, al igual que las técnicas presentadas en el capítulo 3, este análisis es un análisis *Data Flow* de los tipos *forward* y *may*.

### Resolución de referencias con descubrimiento de posiciones de memoria

Antes de pasar al segundo conjunto de reglas es necesario describir un proceso que se presenta al analizar un programa, y es el "descubrimiento" de posiciones de memoria. Este descubrimiento es análogo a lo que se realiza en las técnicas de *Shape Analysis* [SRW99].

Si se observa el programa de la figura 5.4 se ve que la abstracción del *heap* antes de la sentencia "`2:`" es:

$(\{tr_{null}^{Nodo} \rightarrow \{this, tmp\}\}, \emptyset, \emptyset)))$

```
class Nodo {

    Nodo next;

    public void recorrerUno() {
1:      Nodo tmp = this;
2:      tmp = tmp.next;
    }
}
```

Figura 5.4: Programa para recorrer el primer elemento de una lista encadenada.

Al momento de analizar la sentencia "`2:`" será necesario resolver la referencia *tmp.next* para hacer que la variable *tmp* apunte a los mismos lugares que *tmp.next*. Para ver qué posición de memoria es apuntada por *tmp.next*, primero es necesario resolver la referencia *tmp*. La resolución de dicha referencia arroja por resultado las posiciones de memoria $tr_{null}^{Nodo}$. Luego hay que resolver las referencia $tr_{null}^{Nodo}.next$. Al intentar hacerlo se observa que no existe ninguna posición de memoria en la abstracción del *heap* para esa referencia. Lo que ocurre es que $tr_{null}^{Nodo}.next$ hace referencia a un campo del objeto de tipo "TR" que se utiliza para representar simbólicamente el objeto *this*. Por *default*, los campos de tipo referencia del objeto *this* no se representan en la abstracción a menos que sea necesario, como ocurre al analizar la sentencia "`2:`". En este caso se dice que se "descubre" un campo referencia y la abstracción del *heap* queda de la siguiente forma:

$(\{tr_{null}^{Nodo} \rightarrow \{this, tmp, tr_{null}^{Nodo}.next\}, dt_2^{Nodo} \rightarrow \{tr_{null}^{Nodo}.next, this, tmp\}\}$

$\{tr_{null}^{Nodo}.next\}, \emptyset)))$

Notar la necesidad de agregar la referencia $tr_{null}^{Nodo}.next$ al conjunto de referencias nulas, ya que no es posible asegurar que la referencia efectivamente apunte a una posición

de memoria. Además, es necesario que la nueva posición de memoria también sea apuntada las referencias correspondientes, ya que se deben modelar todas las alternativas que respeten la relación *esAsignableDesde*.

Una situación análoga se presenta cuando se "descubren" atributos de argumentos simbólicos objeto (SA) y cuando se "descubren" atributos de objetos que a su vez también han sido descubiertos.

Una situación particular que se presenta es que si, por ejemplo, se considera el programa de la figura 5.5, se observa que la sentencia "`1:`" da puede dar lugar a múltiples descubrimientos. Cada una de esas posiciones de memoria descubiertas estará asociada a la misma sentencia, pero poseerá un número de serie diferente.

```
class Nodo {

    Nodo next;

    public void recorrer() {
        Nodo tmp = this;
        while (tmp != null) {
 1:         tmp = tmp.next;
        }
        return;
    }
}
```

Figura 5.5: Programa para recorrer una lista encadenada.

Más adelante en la sección 5.3.2 se muestra como evitar que una serie de descubrimientos de posiciones de memoria, como la presente en la figura 5.5, provoque que la abstracción del *heap* crezca indefinidamente.

### Segundo conjunto de reglas

A continuación se muestra el segundo conjunto de reglas, las que relacionan la restricción después de ejecutar una sentencia con la restricción antes de ejecutarla. Las reglas se pueden considerar como una función que recibe como argumento la restricción antes de ejecutar la sentencia ($Etq_i^{antes}$) y la sentencia correspondiente a la etiqueta $Etq_i$. Dicha función se denota $f(Etq_i^{antes}, s)$ y es la función de transferencia del análisis *Data Flow*.

- $R_3^{oo}$. si s = `if ( ` **ExpresiónBooleana** `) goto` **Etiqueta**
  $Etq_i^{después} = Etq_i^{antes}$
- $R_4^{oo}$. si s = `goto` **Etiqueta**
  $Etq_i^{después} = Etq_i^{antes}$
- $R_5^{oo}$. si s = `return`
  $Etq_i^{después} = Etq_i^{antes}$
- $R_6^{oo}$. si s = `return` **OperandoEntero**
  $H(Etq_i^{después}) = H(Etq_i^{antes})$

$D(Etq_i^{después}) =$ aplicar la regla $R_4^P$ de la sección 3.2.2

- $R_7^{oo}$. si s = `return` **OperandoReferencia** $\wedge$ **OperandoReferencia** $= var$
  $H(Etq_i^{después}) = addAlias(H(Etq_i^{antes}), retValue, var, false))$
  $D(Etq_i^{después}) = D(Etq_i^{antes})$
- $R_8^{oo}$. si s = `return` **OperandoReferencia** $\wedge$ **OperandoReferencia** = `null`
  $H(Etq_i^{después}) = addNullAlias(H(Etq_i^{antes}), retValue))$
  $D(Etq_i^{después}) = D(Etq_i^{antes})$
- $R_9^{oo}$. si s $= var_1$ `=` **ExpresiónZ** $\wedge$ **ExpresiónZ** $\neq var_2.var_3 \wedge$ **ExpresiónZ** $\neq$ **ExpresiónZNoLineal** $\wedge$ **ExpresiónZ** $\neq$ **Invocación**
  $H(Etq_i^{después}) = H(Etq_i^{antes})$
  $D(Etq_i^{después}) =$ aplicar la regla $R_5^P$ de la sección 3.2.2
- $R_{10}^{oo}$. si s $= var_1$ `=` **ExpresiónZ** $\wedge$ **ExpresiónZ** $= var_2.var_3$
  $H(Etq_i^{después}) = H(Etq_i^{antes})$

  Para resolver la parte del dominio deben realizarse lo siguiente:

  - realizar la resolución de referencias de la expresión $var_2.var_3$ (ver sección 5.3.1). El conjunto $C$ contendrá el resultado de dicha resolución (o sea el conjunto de variables a las que hace referencia la expresión $var_2.var_3$)
  - Para cada elemento $c_j$ del conjunto $C$ realizar la substitución de $c_j$ por $var_2.var_3$ en la expresión $var_1 = var_2.var_3$ dando lugar a $\sharp(C)$ expresiones (una expresión para cada elemento de $C$). Se llama $R$ al conjunto que contiene cada una de las expresiones que se obtiene luego de las substituciones.
  - Para cada expresión de $R$ aplicar la regla $R_5^P$ de la sección 3.2.2. Esta operación da lugar a una cantidad de $\sharp(R)$ ecuaciones y nuevas variables que se denotan $Etq_{i_j}^{después}$
  - Finalmente se tiene que:

$$D(Etq_i^{después}) = \bigcup_{j=1}^{j \leq \sharp(R)} Etq_{i_j}^{después}$$

- $R_{11}^{oo}$. si s $= var$ `=` **ExpresiónZNoLineal**
  $H(Etq_i^{después}) = H(Etq_i^{antes})$
  $D(Etq_i^{después}) =$ aplicar la regla $R_6^P$ de la sección 3.2.2
- $R_{12}^{oo}$. si s = **AsignaciónEntera** $\wedge$ **AsignaciónEntera** $= var_1.var_2$`=`$var_3$
  $H(Etq_i^{después}) = H(Etq_i^{antes})$

  Para resolver la parte del dominio deben realizarse lo siguiente:

  - realizar la resolución de referencias de la expresión $var_1.var_2$ (ver sección 5.3.1). El conjunto $C$ contendrá el resultado de dicha resolución (o sea el conjunto de variables a las que hace referencia la expresión $var_1.var_2$)

- Para cada elemento $c_j$ del conjunto $C$ realizar la substitución de $c_j$ por $var_1.var_2$ en la expresión $var_1.var_2 = var_3$ dando lugar a $\sharp(C)$ expresiones (una expresión para cada elemento de $C$). Se llama $R$ al conjunto que contiene cada una de las expresiones que se obtiene luego de las substituciones.
- Para cada expresión de $R$ aplicar la regla $R_5^P$ de la sección 3.2.2. Esta operación da lugar a una cantidad de $\sharp(R)$ ecuaciones y nuevas variables que se denotan $Etq_{i_j}^{después}$
- Luego se tiene que:

$$D(Etq_i^{después}) = \bigcup_{j=1}^{j \leq \sharp(R)} D(Etq_{i_j}^{después})$$

Y como último detalle hay que considerar que si al realizar la resolución de referencias se encuentra que alguna de las posiciones de memoria a las que apunta $var_1$ está incluida dentro del conjunto de *weak updates* de la abstracción del *heap*, es necesario definir $D(Etq_i^{después})$ de la siguiente manera

$$D(Etq_i^{después}) = \bigcup_{j=1}^{j \leq \sharp(R)} D(Etq_{i_j}^{después}) \cup D(Etq_i^{antes})$$

de esta forma al estar en presencia de un *weak update* se refleja la necesidad de conservar la información anterior realizando la conjunción con $D(Etq_i^{antes})$.

- $R_{13}^{oo}$. si s $= var$ `= new` $nc_1$

  $H(Etq_i^{después}) = addNewEntry(H(Etq_i^{antes}), cs_i^{nc_1}, var)$

  $D(Etq_i^{después}) = D(Etq_i^{antes})$

  A la abstracción del *heap* original se la agrega una nueva posición de memoria apuntada por $ref$. El dominio se conserva intacto.

- $R_{14}^{oo}$. si s $= var$ `= null`

  $H(Etq_i^{después}) = addNullAlias(H(Etq_i^{antes}), var)$

  $D(Etq_i^{después}) = D(Etq_i^{antes})$

  A la abstracción del *heap* original se la agrega $ref$ al conjunto de referencias nulas.

- $R_{15}^{oo}$. si s $= var_1$ `=` **ExpresiónReferencia** $\wedge$ **ExpresiónReferencia** $= var_2$

  $H(Etq_i^{después}) = addAlias(H(Etq_i^{antes}), var_1, var_2, false)$

  $D(Etq_i^{después}) = D(Etq_i^{antes})$

  En la abstracción del *heap* original se hace que $var_1$ apunte a las mismas posiciones de memoria que $var_2$. *Weak update* es falso porque se trata de una asignación que efectivamente se realiza.

- $R_{16}^{oo}$. si s $= var_1$ `=` **ExpresiónReferencia** $\wedge$ **ExpresiónReferencia** $= var_2.var_3$

  Obtener el conjunto $C$ de resolución de referencias con descubrimiento de posiciones de memoria de $var_2.var_3$.

Luego realizar iterativamente, para cada elemento $c_j$ de $C$

$$X_j = addAlias(X_{j-1}, var_1, c_j, weak_j)$$

donde $X_0 = H(Etq_i^{antes})$.

Así se obtiene:

$H(Etq_i^{después}) = X_{\sharp(C)}$

El valor de $weak_1$ es false, para hacer que toda información previa que se tenga de $var_1$ se pierda, pero $weak_j = true$ para $1 < j \leq \sharp(C)$ para conservar todas las posibles asignaciones que se derivan de $var_2.var_3$.

La parte del dominio permanece intacta

$D(Etq_i^{después}) = D(Etq_i^{antes})$

- $R_{17}^{oo}$. si $\mathbf{s} = \mathbf{AsignaciónReferencia} \wedge \mathbf{AsignaciónReferencia} = var_1.var_2$ = $var_3$ Obtener el conjunto $C$ de resolución de referencias de $var_1.var_2$.

  Si la cantidad de elementos de $C$ es uno, $\sharp(C) = 1$, y el único elemento se denota $c_1$

  $$H(Etq_i^{después}) = addAlias(H(Etq_i^{antes}), c_1, var_3, weak)$$

  donde el valor de weak es *true* si la posición de memoria apuntada por $var_1$ está dentro del conjunto de *weak updates* de la abstracción del *heap*, o el valor es *false* en caso contrario.

  Si $\sharp(C) > 1$ se debe realizar iterativamente, para cada elemento $c_j$ de $C$

  $$X_j = addAlias(X_{j-1}, c_j, var_3, true)$$

  donde $X_0 = H(Etq_i^{antes})$.

  Así se obtiene:

  $H(Etq_i^{después}) = X_{\sharp(C)}$

  La parte del dominio permanece intacta

  $D(Etq_i^{después}) = D(Etq_i^{antes})$

A continuación se muestra, a modo de ejemplo, como se aplican algunas de las reglas definidas. El programa a considerar es el siguiente:

```
class A { int f; A ref; }

class B extends A {
    m1(int p1) {
        A a;
        if (p1 > 0) {
            a = new A();
        } else {
            a = new B();
        }
  1:    a.f = 1;
  2:    A a2 = new A();
  3:    a.ref = a2;
    }
}
```

El gráfico de la figura 5.6 muestra un esquema de la abstracción del *heap* cuando se aplica la regla $R_2^{oo}$ que corresponde al punto del programa $1^{antes}$

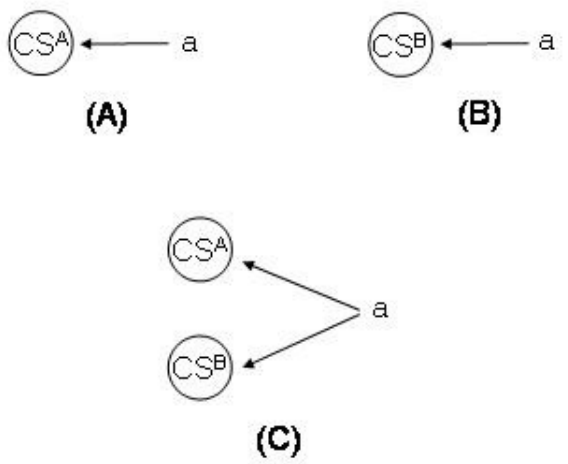


Figura 5.6: (A) diagrama de la abstracción del *heap* correspondiente al *branch* afirmativo del `if`. (B) diagrama de la abstracción del *heap* correspondiente al *branch* negativo del `if`. (C) diagrama de la abstracción del *heap* correspondiente a la unión de ambos casos.

Al analizar la sentencia "`1:`" se debe aplicar la regla $R_{12}^{oo}$. Para esto es necesario resolver la referencia $a.f$ dando lugar al conjunto de variables $\{ cs^A.f, cs^B.f \}$. Luego las substituciones dan lugar a dos expresiones: $cs^A.f = 1$ y $cs^B.f = 1$. Finalmente la parte correspondiente al dominio de la restricción para el punto del programa $1^{después}$ es $\{ cs^A.f = 1 \} \cup \{ cs^B.f = 1 \}$

Para analizar el punto del programa $2^{después}$ es necesario aplicar la regla $R_{13}^{oo}$. En la figura 5.7 se muestra el diagrama correspondiente a la abstracción del *heap*.

Finalmente, para analizar la restricción correspondiente al punto del programa $3^{después}$ es necesario aplicar la regla $R_{13}^{oo}$. Primero se resuelve la referencia $a.ref$ dando lugar al conjunto de variables $\{ cs^A.ref, cs^B.ref \}$. Luego cada una de estas variables debe ser

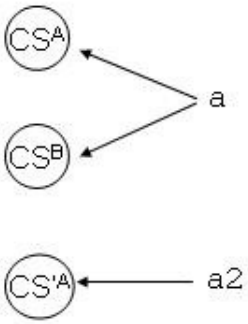


Figura 5.7: Diagrama de la abstracción del *heap* correspondiente al punto del programa $2^{después}$.

un alias de la referencia $a2$. En la figura 5.8 se muestra un diagrama de como queda la abstracción del *heap* para el punto del programa $3^{después}$.

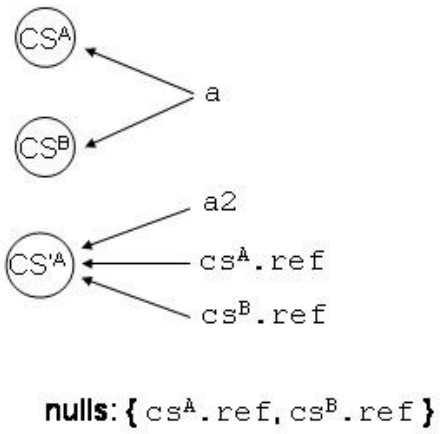


Figura 5.8: Diagrama de la abstracción del *heap* correspondiente al punto del programa $3^{después}$.

En las reglas definidas anteriormente se omitieron las invocaciones a métodos tanto en las expresiones de números enteros como en las expresiones de referencias. Las invocaciones a métodos en esta sección se analizarán de acuerdo con la técnica de *inlining*. No se realizará análisis de invocaciones de métodos en forma simbólica para programas con características de orientación a objetos, sino que se presenta como trabajo a futuro (ver 6.1).

Tomando como base la técnica de análisis interprocedural mediante *inlining* presentada en la sección 3.3.2 sólo deben realizarse algunas modificaciones para adaptarla a programas con características de orientación a objetos.

Lo primero que debe considerarse es que además de necesitar el *binding* de los argumentos de tipo `int` con los meta-argumentos, y de estos últimos con los parámetros formales (al igual que en programas de números enteros), es necesario hacer que los parámetros formales de tipo referencia sean alias de los argumentos correspondientes. Además, debe tenerse en cuenta que una expresión entera puede, al resolverse, dar por resultado varios valores, por lo cual el dominio que actúa como precondición debe considerar todos los casos posibles mediante disyunciones.

Un segundo aspecto a considerar es que en algunas situaciones no es posible resolver estáticamente el *binding* del nombre del método invocado con el cuerpo de un método particular. Por ejemplo, si se considera el siguiente programa de la figura 5.9,

```
class A {
    int f;

    init() {
        this.f = 0;
    }

    m1(A pA) {
        pA.init();
    }
}
```

```
class B extends A {

    init() {
        this.f = 1;
    }

}
```

Figura 5.9: Ejemplo para resolución de *binding* dinámico de invocaciones.

al analizar el método $m1$ la posición de memoria apuntada por $pA$ es de la variante "SA", o sea es una posición de memoria simbólica, con lo cual no es posible determinar cual es el cuerpo del método *init*() que efectivamente debe considerarse en el análisis. Ante esta situación sólo pueden realizarse los análisis de ambas posibilidades y luego unir los resultados. Esta necesidad de considerar todas las opciones a las que se puede resolver el *binding* dinámico hace que la técnica presentada en este trabajo sólo sea útil en caso de *sistemas cerrados*. Un sistema cerrado es aquél en el cual todas las clases están determinadas en tiempo de compilación y no existe carga dinámica de clases. Por el contrario un sistema abierto es aquél para el cual no es posible determinar en tiempo de compilación las clases que intervienen ya que existe la posibilidad de carga dinámica de clases.

Las invocaciones sobre objetos que responden a la variante "CS" sí pueden ser resueltas estáticamente porque se conoce exactamente de qué clase es instancia el objeto en cuestión, pero por el contrario las invocaciones sobre objetos que responden a las variantes "TR", "SA", "DA" y "DT" necesitan que se contemplen todas las posibilidades de resolución del *binding*.

Otra de las cuestiones a tener en cuenta al analizar invocaciones en programas con características de orientación a objetos mediante *inlining* es que la variable distinguida *this* debe resolverse cuidadosamente de acuerdo con el contexto en el que se encuentra. Para esto es necesario que al empezar a analizar un método invocado se modifique la abstracción del *heap* para que la variable *this* no apunte a una posición de memoria de la variante "TR", sino que apunte a la posición de memoria que corresponda. Por ejemplo, al analizar el método $m1$ del programa de la figura 5.9 dentro de los métodos *init*() la variable distinguida *this* apunta a la posición de memoria de la variante "SA" apuntada por el parámetro formal $pA$. Ante la posibilidad de múltiples invocaciones es necesario

conservar el estado de la variable *this* en una estructura de pila para poder restaurar el valor correspondiente a cada contexto.

Una última diferencia con la técnica utilizada en programas de números enteros es que el valor de retorno del método puede ser, además de una expresión tipo `int`, una referencia o ningún valor (métodos *void*). En el primer caso se hace lo mismo que en el análisis de programas de números enteros, o sea se realiza el *binding* de la variable que se está definiendo con el valor de retorno del método invocado, considerando en este caso que la resolución de referencias puede dar lugar a múltiples definiciones. En el segundo caso, se debe hacer que la referencia que se está definiendo sea alias de la referencia que actúa como valor de retorno del método que se invoca. En el tercer caso simplemente la post-condición del método será el valor de la incógnita después de ejecutar la invocación.

### Convergencia del *heap*

El descubrimiento de posiciones de memoria que se presenta en la resolución de referencias trae aparejado el siguiente inconveniente. Si se considera el siguiente programa en una hipotética sintaxis concreta:

```
class A {

    A   next;
    int f;

    init(A pA) {
        while (pA.next != null) {
1:          pA = pA.next;
2:          pA.f = 0;
        }
    }
}
```

al analizar el método *init* cada iteración descubre una nueva posición de memoria de la variante "DA" y la cadena creciente que se genera en la búsqueda del punto fijo diverge infinitamente.

Para evitar esta situación es necesario, al igual que se hace al analizar programas de números enteros, realizar algún tipo de *widening* que evite el infinito descubrimiento de posiciones de memoria.

Una de las técnicas utilizadas en *shape analysis* es considerar que cada sentencia del programa puede dar lugar a una única posición de memoria. De esta forma la sentencia "`1:`" dará lugar al descubrimiento de una única posición de memoria y la cadena ascendente se estabilizará.

Para asegurar la convergencia en el *heap* se utiliza el siguiente operador:

$merge(AHeap) : AHeap$

ENTRADA: $aHeap_1 = (heapMap_1, nulls_1, weakUpdates_1)$

SALIDA: $aHeap_{ret} = (heapMap_{ret}, nulls_{ret}, weakUpdates_{ret})$

ALGORITMO: $aHeap_{ret} = aHeap_1$;
para cada $ml_i \in Dom(heapMap_{ret})$
si $ml_i$ es de la variante "CS" o "DA" o "DT"
$par = findPair(heapMap_{ret}, ml_i)$
si $(par \neq null)$
$tmp = heapMap_{ret}(ml_i) \cup heapMap_{ret}(par)$;
$heapMap_{ret} = heapMap_{ret} \backslash \{(ml_i \rightarrow heapMap_{ret}(ml_i))\} \cup$
$\{\ (ml_i \rightarrow tmp)\}$;
$heapMap_{ret} = heapMap_{ret} \backslash \{(par \rightarrow heapMap_{ret}(par))$;
$weakUpdates_{ret} = weakUpdates_{ret} \cup \{ml_i\}$;

Este procedimiento itera sobre cada posición de memoria de la abstracción del *heap* y para cada una de ellas busca su "par". Una posición de memoria es "par" de otra si ambas están asociadas a la misma sentencia (etiqueta) del programa y tienen números de serie diferente. Si se encuentra el par se unen las referencias de ambas posiciones de memoria para que apunten a una sola y se elimina la posición de memoria con el número de serie más grande. Además la posición de memoria que ha quedado debe agregarse a la lista de posiciones sobre las que se realiza *weak update*. Esto se debe a que una sola posición de memoria representa varias potenciales en tiempo de ejecución y en consecuencia no se deberá perder la información que exista asociada a ella con cada nueva definición.

Se deja como trabajo a futuro demostrar que este operador es efectivamente un operador de *widening*. De todas formas, se ha comprobado su efectividad con la implementación descripta en la sección 5.7

Finalmente, para realizar el *widening* se combina el operador presentado en la sección 3.4.1, para la componente de la restricción que es un dominio de poliedros, y el operador *merge* para la componente de la restricción que es una instancia de *AHeap*.

## 5.4. Características avanzadas

En esta sección se mencionan algunas de las características de la técnica que son consideradas adicionales. No se realiza una descripción en profundidad, sino que se mencionan las principales ideas que fueron tenidas en cuenta para la incorporación de las correspondientes características al análisis.

### 5.4.1. *Arrays*

La mayoría de los lenguajes de programación provee algún mecanismo de manipulación de *arrays*. Es por esto que resulta interesante que la técnica presentada considere los *arrays* dentro del análisis.

Lo primero que resulta necesario es extender el lenguaje para soportar *arrays*. Las principales extensiones al lenguaje de la sección 5.1 son:

ExpresiónReferencia → `new` *nombreClase* `[` OperandoEntero `]`
ExpresiónReferencia → `new int [` OperandoEntero `]`
ExpresiónReferencia → *nombreVariable*`[`OperandoEntero`]`

ExpresiónZ $\rightarrow$ *nombreVariable*[OperandoEntero]

AsignaciónEntera $\rightarrow$ *nombreVariable*$_1$[OperandoEntero] = *nombreVariable*$_2$

AsignaciónReferencia$\rightarrow$ *nombreVariable*$_1$[OperandoEntero] = *nombreVariable*$_2$

Luego es necesario incorporar una nueva variante para las posiciones de memoria que representan *arrays*. Dicha variante será denotada "AR".

La manipulación de *arrays* cuando se realiza análisis estático es intrínsecamente compleja debido a que en muchos casos la dimensión del *array* sólo queda determinada en tiempo de ejecución. Una forma habitual de abstraer dicha complejidad es considerar que los arrays son todos de dimensión uno. De esta forma claramente se pierde precisión, pero se logra manipular estáticamente objetos cuyas principales características quedan determinadas en tiempo de ejecución.

Junto con la asunción de que los arrays tienen dimensión uno es necesario agregar cada posición de memoria de la variate "AR" al conjunto de *weak updates*. De esta forma cada potencial modificación de los valores del *array* quedará reflejada en las restricciones.

Para analizar los *arrays* simplemente se considera que un *array* es un objeto con un único atributo del tipo correspondiente y se aplican las reglas presentadas anteriormente. De esta forma, por ejemplo, las sentencias:

```
a = new int[count];

a[index] = 3;
```

serían análogas a las sentencias hipotéticas:

```
a = new MiIntArray;

a.field = 3;
```

donde *MiIntArray* es:

```
class MiIntArray {
    int field;
}
```

Además del análisis de los *arrays* descripto anteriormente es posible realizar una pequeña modificación para incrementar la precisión del análisis. Dicha modificación consiste en generar para cada *array* una variable que permita predicar sobre la longitud del *array*. Por ejemplo, al analizar la sentencia:

```
a = new int[count];
```

se genera la variable $ar^{int}.length$ y se agrega al dominio la restricción $\{\ ar^{int}.length = count \wedge ar^{int}_{etq}.length > 0\ \}$, ya que por ejemplo, el lenguaje de programación Java no permite la creación de *arrays* de longitud negativa.

Más interesante aun resulta, por ejemplo, cuando se analiza la sentencia:

```
a[index] = 3;
```

agregar al dominio la restricción $\{\ 0 \leq index < ar^{int}.length\ \}$, lo que en definitiva reflejará la relación $0 \leq index < count$, y permitirá detectar estáticamente posibles, o efectivas, indexaciones a *arrays* fuera de los rangos permitidos.

### 5.4.2. Elementos estáticos

Es común en lenguajes de programación con características de orientación a objetos que exista la posibilidad de declarar como estáticos tanto métodos como atributos de una clase. En este trabajo la semántica del calificador de un elemento como estático será la que existe en el lenguaje de programación Java.

Las principales extensiones al lenguaje para poder considerar elementos estáticos son:

DeclaraciónEntera → `static int` $nombreVariable$

DeclaraciónReferencia → `static` $nombreClase\ nombreVariable$

Método → `static` $nombreMétodo$ `() {` SentenciaEtiquetada `}`
Método → `static` $nombreMétodo$ `(` Parámetros `) {` SentenciaEtiquetada `}`

ExpresiónZ → $nombreClase.nombreVariable$

ExpresiónReferencia → $nombreClase.nombreVariable$

DefiniciónEntera → $nombreClase.nombreVariable_1$ `=` $nombreVariable_2$

DefiniciónReferencia → $nombreClase.nombreVariable_1$ `=` $nombreVariable_2$

Invocación → $nombreClase.nombreMétodo$`()`
Invocación → $nombreClase.nombreMétodo$`(`Argumentos`)`

La invocación de métodos estáticos prácticamente no requiere de ninguna modificación. Además la invocación estática sí puede resolverse en tiempo de compilación, así que no existe ambigüedad para determinar el cuerpo del método que se debe analizar en una invocación estática. La única diferencia es que no debe realizarse la modificación de la referencia distinguida *this*, ya que en un contexto estático nunca puede haber referencias al objeto *this*.

La utilización de una variable estática de tipo entero tampoco presenta ningún inconveniente. Simplemente se debe considerar que si se intenta resolver una expresión de la forma $nombreClase.nombreVariable$ en realidad $nombreClase$ no es una referencia a ninguna posición de memoria de la abstracción del *heap*. Es por esto que no es necesario realizar ninguna resolución ya que siempre hace referencia al mismo elemento y se puede denotar directamente $nombreClase.nombreVariable$.

La utilización de una referencia estática debe ser tratada con más cuidado, ya que no necesariamente se conoce la posición de memoria a la cual la referencia apunta. Por ejemplo, en el siguiente programa

```
class C {

    static A sA;
    int f;

    m1(A pA) {
1:      pA.f = 1;
2:      A other = A.sa;
3:      oher.f = 2;
    }
}
```

al analizar la sentencia "`2:`" se encuentra que la referencia *other* debe apuntar a la misma posición de memoria que lo hace la referencia estática *sa* de la clase $A$.

Este problema es análogo al que se presenta al analizar la sentencia "`3:`" de la figura 5.2, y se resuelve mediante la resolución de referencias con descubrimiento de posiciones de memoria. En el caso de las posiciones descubiertas apuntadas por referencias estáticas se utilizará una nueva variante de posiciones de memoria denotada "SR" (*static reference*). Al igual que con el descubrimiento de posiciones de memoria que son campos de argumentos simbólicos, también es necesario realizar la "clausura" de las referencias con respecto a la relación $esAsignableDesde$. De esta forma la abstracción del *heap* después de la sentencia "`2:`" es:

$(\{sr_2^A \rightarrow \{A.sa, pA, other\}, sa_{null}^A \rightarrow \{pA, A.sa, other\}, tr_{null}^C \rightarrow \{this\}\},$
$\{pA, A.sa, other\}, \emptyset)))$

### 5.4.3. *Predicate abstraction*

Si se analiza el método $m1$ del programa de la figura 5.10 puede observarse lo siguiente.

La restricción antes de la sentencia "`7:`", denotada $7^{antes}$, es:

```
 class A {                                  class B extends A {
     int f;                                     ...
                                            }
     m1(int p1) {
1:       A miA = null;
2:       int c = 0;
         if (p1 > 0) {
3:           miA = new A;
4:           c = 1;
         } else {
5:           miA = new B;
6:           c = 2;
         }
7:       miA.f = c;
     }
 }
```

Figura 5.10: Ejemplo para *predicate abstraction*.

$$(\{p1 = p1_0 \wedge c = 1 \wedge p1_0 \geq 1\} \vee \{p1 = p1_0 \wedge c = 2 \wedge p1_0 \leq 0\},$$
$$(\{cs_3^A \rightarrow \{miA\}, cs_5^B \rightarrow \{miA\}, tr_{null}^A \rightarrow \{this\}\}, \emptyset, \emptyset))$$

Al analizar la sentencia "7:" se debe resolver la variable $miA.f$. La resolución de dicha variable da como resultado el conjunto $\{cs_3^A.f, cs_5^B.f\}$. Luego, la restricción después de la sentencia "7:", denotada $7^{después}$, puede representarse como:

$$7^{después} = (\{(7^{antes} \cap \{cs_3^A.f = c\}) \cup (7^{antes} \cap \{cs_5^B.f = c\})\},$$
$$(\{cs_3^A \rightarrow \{miA\}, cs_5^B \rightarrow \{miA\}, tr_{null}^A \rightarrow \{this\}\}, \emptyset, \emptyset))^2$$

Si se expande la expresión que corresponde al dominio de $7^{después}$ se obtiene:

$$\{p1 = p1_0 \wedge c = 1 \wedge p1_0 \geq 1 \wedge cs_3^A.f = c\} \vee$$
$$\{p1 = p1_0 \wedge c = 2 \wedge p1_0 \leq 0 \wedge cs_3^A.f = c\} \vee$$
$$\{p1 = p1_0 \wedge c = 1 \wedge p1_0 \geq 1 \wedge cs_5^B.f = c\} \vee$$
$$\{p1 = p1_0 \wedge c = 2 \wedge p1_0 \leq 0 \wedge cs_5^B.f = c\}$$

Si bien este dominio resulta correcto, hay dos componentes de la disyunción que no tiene sentido que figuren en el mismo. Ellos son el segundo y el tercero. En el segundo se ve la relación $cs_3^A.f = c$, que corresponde al *branch* positivo del `if`, y la relación $c = 2$, que corresponde al *branch* negativo, lo que no puede suceder en ninguna ejecución real del programa. En la tercera componente de la disyunción se presenta una inconsistencia análoga.

El problema de la falta de precisión radica en que, por un lado, la abstracción del *heap* de la restricción antes de la sentencia "7:" refleja que la variable $miA$ puede apuntar a dos posiciones de memoria distintas, y por otro lado, el dominio refleja, mediante

[2] en esta fórmula se abstrae la complejidad real de la regla donde es necesario incorporar una variable temporal, luego realizar el *binding* de dicha variable con la variable que se define y finalmente eliminar la variable temporal.

la disyunción, las dos alternativas del flujo del programa; pero ningún elemento de la restricción en general permite relacionar las disyunciones en el dominio o en el *heap* de alguna manera. Debido a esto es necesario, en algún sentido, realizar la combinatoria de ambas disyunciones dando lugar a la pérdida de información descripta.

Para solucionar este problema se puede utilizar *predicate abstraction* (o abstracción de predicados). La idea principal es definir un predicado:

$$apuntaA(Referencia, MemoryLocation)$$

que representa el concepto de que una variable efectivamente apunta a una posición de memoria particular. Por ejemplo, después de la sentencia "`3:`" del programa de la figura 5.10 es válido el predicado:

$$apuntaA(miA, cs_3^A)$$

Luego el predicado puede abstraerse en una variable entera que se denota $miA@cs_3^A$, y dicha variable entera puede ser incorporada a un dominio de poliedros convexos cerrados de forma tal que si su valor es 0 el predicado es falso, y si su valor es 1 el predicado es verdadero.

De esta forma la restricción antes de la sentencia "`7:`" quedaría:

$$\begin{array}{l}(\{p1 = p1_0 \wedge c = 1 \wedge p1' \geq 1 \wedge miA@cs_3^A = 1\} \vee \\ \{p1 = p1_0 \wedge c = 2 \wedge p1' \leq 0 \wedge miA@cs_5^B = 1\}, \\ (\{cs_3^A \rightarrow \{miA\}, cs_5^B \rightarrow \{miA\}, tr_{null}^A \rightarrow \{this\}\}, \emptyset, \emptyset))\end{array}$$

Ahora sí existe una relación entre la información de la abstracción del *heap* y el dominio de poliedros.

Para aprovechar la información que se recolectó es necesario que al momento de resolver las referencias se genere un conjunto de restricciones que relacionen a cada resultado de la resolución con la abstracción de predicados realizada. En el caso del ejemplo, al resolver la expresión $miA.f$, se obtiene:

$$\begin{array}{lcl} cs_3^A.f & \rightarrow & \{\ miA@cs_3^A = 1 \wedge miA@cs_5^B = 0\ \} \\ cs_5^B.f & \rightarrow & \{\ miA@cs_3^A = 0 \wedge miA@cs_5^B = 1\ \} \end{array}$$

Luego, al utilizar cada solución en particular se deben agregar a la restricción las relaciones. En el caso del ejemplo

$$\begin{array}{lcl} D(7^{después}) & = & \{7^{antes} \cap \{cs_3^A.f = c \wedge miA@cs_3^A = 1 \wedge miA@cs_5^B = 0\}\} \cup \\ & & \{7^{antes} \cap \{cs_5^B.f = c \wedge miA@cs_3^A = 0 \wedge miA@cs_5^B = 1\}\} \end{array}$$

y al expandir la expresión se obtiene:

$$\begin{array}{l}\{p1 = p1_0 \wedge c = 1 \wedge p1_0 \geq 1 \wedge miA@cs_3^A = 1 \wedge cs_3^A.f = c \wedge miA@cs_3^A = 1 \wedge miA@cs_5^B = 0\} \vee \\ \{p1 = p1_0 \wedge c = 2 \wedge p1_0 \leq 0 \wedge miA@cs_5^B = 1 \wedge cs_3^A.f = c \wedge miA@cs_3^A = 1 \wedge miA@cs_5^B = 0\} \vee \\ \{p1 = p1_0 \wedge c = 1 \wedge p1_0 \geq 1 \wedge miA@cs_3^A = 1 \wedge cs_5^B.f = c \wedge miA@cs_3^A = 0 \wedge miA@cs_5^B = 1\} \vee \\ \{p1 = p1_0 \wedge c = 2 \wedge p1_0 \leq 0 \wedge miA@cs_5^B = 1 \wedge cs_5^B.f = c \wedge miA@cs_3^A = 0 \wedge miA@cs_5^B = 1\}\end{array}$$

donde la segunda y tercera componente del dominio son contradicciones, por lo tanto el resultado definitivo es:

$$\{p1 = p1' \wedge c = 1 \wedge p1' \geq 1 \wedge miA@cs_3^A = 1 \wedge cs_3^A.f = c \wedge miA@cs_5^B = 0\} \cup$$
$$\{p1 = p1' \wedge c = 2 \wedge p1' \leq 0 \wedge miA@cs_5^B = 1 \wedge cs_5^B.f = c \wedge miA@cs_3^A = 0\}$$

De esta forma se eliminan del dominio las posibilidades que en tiempo de ejecución no son factibles y en conclusión se logra aumentar la precisión del análisis.

## 5.5. *Feedback* entre el dominio y la abstracción del *heap*

Una de las características más interesantes de la técnica presentada es que el análisis de las restricciones lineales se realiza conjuntamente con el análisis de abstracción del *heap*. Esta "simultaneidad" permite que ambos aspectos del análisis se nutran de la información del otro dando lugar a un incremento de la precisión.

Por un lado, se tiene que el análisis de abstracción del *heap* utiliza la información que proveen los dominios cuando, en los puntos en los que se deben unir varios flujos alternativos del programa, se ignora la información provista por los *branches* que representan "código muerto". Por ejemplo si se considera el siguiente programa:

```
 class A {

     int f;

     m1(int p1) {
1:       A a1 = new A;
2:       A a2 = null;
3:       A a3 = null;
         if (p1 > 5) {
4:           a2 = new A;
             if (p1 > 0) {
5:               a3 = a1;
             } else {
6:               a3 = a2;
             }
7:           a3.f = 1;
         }
     }
 }
```

antes de la sentencia "`7:`" la abstracción del *heap* es:

$$(\{cs_1^A \rightarrow \{a1, a3\}, cs_4^A \rightarrow \{a2\}, tr_{null}^A \rightarrow \{this\}\}, \emptyset, \emptyset)$$

donde se ve que la referencia $a3$ únicamente puede apuntar a la misma posición de memoria que la referencia $a1$.

Un analizador estático estándar que realiza "points-to analysis" probablemente no será capaz de descubrir que la sentencia "`6:`" es en realidad código muerto, y deberá conservativamente declarar que la referencia $a3$ puede apuntar tanto al objeto instanciado en la sentencia "`1:`" como al objeto instanciado en la sentencia "`4:`".

La otra faceta del *feedback* entre dominio y abstracción del *heap* es la que se presenta al mejorar la precisión utilizando *predicate abstraction* (ver 5.4.3).

Si la técnica realizara un pre-procesamiento en el cual se utiliza algún "points-to analysis" y luego en una etapa posterior realizara el descubrimiento de relaciones lineales, la pérdida de precisión descripta en la sección dedicada a *predicate abstraction* no podría ser resuelta de la forma en que se lo hizo.

De esta manera queda exhibido el beneficio que se obtiene al realizar ambos análisis de forma simultánea.

## 5.6. Aproximación a la formalización

Si bien la exploración de ideas presentada en las secciones anteriores se basa en la definición de reglas para determinar un análisis de *Data Flow*, aún no es posible afirmar que dicho análisis se enmarca en el *Monotone Framework*. O dicho de otra forma, si bien se presenta un conjunto de reglas que define un sistema de ecuaciones, no es posible afirmar que dicho sistema de ecuaciones tenga solución.

Para poder afirmar que el sistema de ecuaciones que queda determinado por las reglas presentadas tiene solución, es necesario demostrar dos hechos. El primero es que la estructura que se utiliza para abstraer el *heap* ($AHeap$) tiene características de reticulado. El segundo hecho es que la función que se obtiene a partir del sistema de ecuaciones, cuyo punto fijo determina el resultado del análisis, es monótona.

Para demostrar que la estructura $AHeap$ es un reticulado, primero se debe establecer un orden parcial entre los elementos del conjunto. Dicho orden parcial podría obtenerse a partir de la composición de tres relaciones, una para cada uno de los tres elementos que componen la estructura $AHeap$. Para los conjuntos $Nulls$ y $WeakUpdates$ se puede utilizar la relación de inclusión de conjuntos "$\subseteq$". Para el elemento $HeapMap$ podría utilizarse la noción de sub-grafo. Si se observa cuidadosamente la estructura $HeapMap$, puede verse que es un grafo donde los nodos son posiciones de memoria y cada referencia es un eje que tiene como origen y destino posiciones de memoria de la abstracción del *heap*. Un detalle técnico es que es necesario incorporar un nodo (posición de memoria) para que sirva de origen de las referencias que constituyen variables locales, algo así como un nodo que representa el *stack*. A partir de estas nociones parecería posible demostrar que la estructura $AHeap$ es un reticulado.

La completa formalización de las ideas del párrafo anterior y la demostración de que la función que se obtiene a partir del sistema de ecuaciones es monótono exceden los alcances de esta tesis y se plantean como trabajo a futuro (ver sección 6.1). Sin embargo, cabe aclarar que se especula que dichas demostraciones son factibles teniendo en cuenta dos factores. El primero es que la abstracción del *heap* tiene una cantidad de posiciones de memoria (nodos) finita. Esto se debe a que a lo sumo existe un único elemento $MemoryLocation$ por cada sentencia del programa, ya sea porque que realiza alocaciones de memoria (`new`) o presenta descubrimiento de posiciones simbólicas. Esto ocurre aun cuando en tiempo de ejecución la misma sentencia en realidad instancie o acceda a multiples objetos. El segundo factor es que, al analizar un punto del programa, si la sentencia hace apuntar una referencia a determinada posición de memoria, la misma debería permanecer siempre apuntando a dicho lugar en dicho punto del programa; es decir no se eliminan referencias, sino que siempre, a lo sumo, se agregan nuevas. Esto haría que las referencias no oscilen en ese punto del programa, condición necesaria para demostrar la monotonía de la función.

## 5.7. Implementación

A pesar de que la formalización de las ideas presentadas en las secciones anteriores se plantea como trabajo a futuro, sí se realizó una implementación "proof of concept" de dichas ideas.

Para la implementación se extendió el conjunto de reglas existentes en la herramienta descripta en la sección 4.1 de forma tal de considerar las reglas que realizan operaciones con la abstracción del *heap*.

La estructura $(Dominio, AHeap)$, que constituye las incógnitas del análisis, fue implementada reutilizando la implementación de dominio de poliedros convexos cerrados desarrollada anteriormente, y pudo ser fácilmente incorporada la herramienta gracias a la arquitectura de *plug-ins* diseñada.

Cabe destacar que a partir de esta extensión la herramienta es capaz de realizar descubrimiento automático de relaciones lineales entre variables de programas escritos para un subconjunto significativo del lenguaje Java. Quedando fuera del alcance los programas que contienen invocaciones recursivas y la administración de errores mediante el mecanismo de excepciones.

## 5.8. Resultados obtenidos

A continuación se presentan algunos ejemplos que intentan demostrar el potencial y la aplicabilidad de las ideas desarrolladas en este capítulo.

### 5.8.1. Ejemplo 1

En este ejemplo se analiza un programa sencillo que simplemente recorre una lista encadenada. El objetivo del ejemplo es mostrar como el analizador realiza *shape analysis* y abstrae la forma que puede adoptar una instancia de lista encadenada.

El programa a analizar se muestra en la figura 5.11

```
class Nodo {

    Nodo next;

    public void recorrer() {
        Nodo tmp = this;
        while (tmp != null) {
 1:         tmp = tmp.next;
        }
        return;
    }
}
```

Figura 5.11: Programa para recorrer una lista encadenada.

La abstracción del *heap* para el punto del programa $1^{después}$ es:

$$
\begin{array}{lll}
\{ & tr_{null}^{Nodo} & \rightarrow \{ \; this \; \}, \\
 & dt_1^{Nodo} & \rightarrow \{ \; tr_{null}^{Nodo}.next, dt_1^{Nodo}.next, tmp \; \}, \\
\}
\end{array}
$$

$\{ \; tr_{null}^{Nodo}.next, dt_1^{Nodo}.next, tmp \; \}$

$\{ \; dt_1^{Nodo} \; \}$

en la figura 5.12 se muestra en forma gráfica la abstracción del *heap*.

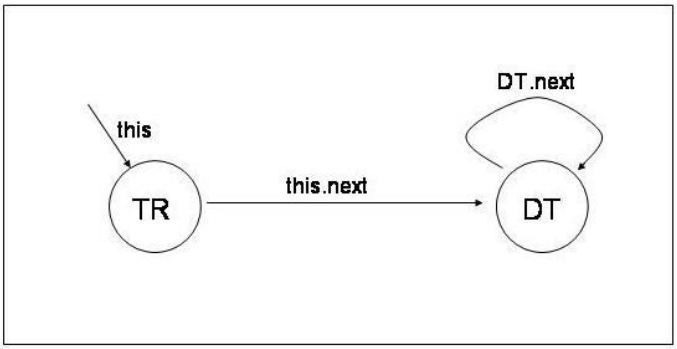


Figura 5.12: Representación gráfica de la abstracción del *heap*.

### 5.8.2. Ejemplo 2

En este ejemplo se mostrará cómo se comporta el análisis ante una situación en la que el *binding* dinámico resulta determinante.

El programa se presenta en la figura 5.13.

```
public int pruebaBindingDinamico(CA pa, int pInt) {

 1:   CA miA = new CA();

      if (pInt > 0) {
          miA.set(1);
      } else {
          miA.set(10);
      }
      pa.set(5);

 2:   int ret = pa.incr(miA.get());
      return ret;
  }
}

class CA {

    int f;

    public void set(int p) {
        f = p;
    }

    public int incr(int p) {
        return  f + p;
    }
}

class CB extends CA {

    public int incr(int p)
    {
        return f + (2*p);
    }
}
```

Figura 5.13: Programa determinado por el *binding* dinámico.

La restricción obtenida en el punto del programa $2^{antes}$ es:

$\{pInt = pInt_0 \wedge cs_1^{CA}.f = 1 \wedge sa_{null}^{CA}.f = 5 \wedge PInt_0 \geq 1\} \cup$

$\{pInt = pInt_0 \wedge cs_1^{CA}.f = 10 \wedge sa_{null}^{CA}.f = 5 \wedge PInt_0 \leq 0\ \}$

la abstracción del *heap* es:

$\{ \quad cs_1^{CA} \quad \rightarrow \quad \{\ miA\ \},$
$\quad\ sa_{null}^{CA} \quad \rightarrow \quad \{\ pA\ \}$
$\}$
$\{\ pA\ \}$
$\emptyset$

La restricción obtenida en el punto del programa $2^{después}$ es:

$\{pInt = pInt_0 \wedge cs_1^{CA}.f = 1 \wedge sa_{null}^{CA}.f = 5 \wedge PInt_0 \geq 1 \wedge ret = 6\} \cup$

$\{pInt = pInt_0 \wedge cs_1^{CA}.f = 10 \wedge sa_{null}^{CA}.f = 5 \wedge PInt_0 \leq 0 \wedge ret = 15\ \} \cup$

$\{pInt = pInt_0 \wedge cs_1^{CA}.f = 1 \wedge sa_{null}^{CA}.f = 5 \wedge PInt_0 \geq 1 \wedge ret = 7\} \cup$

$\{pInt = pInt_0 \wedge cs_1^{CA}.f = 10 \wedge sa_{null}^{CA}.f = 5 \wedge PInt_0 \leq 0 \wedge ret = 25\ \}$

la abstracción del *heap* es la misma que la que se obtiene en el punto del programa $2^{antes}$.

### 5.8.3. Ejemplo 3

Para este ejemplo se tomó el programa de la figura 1 de [BGY05] que se presenta en la figura 5.14

```
    void m0(int mc) {
 1:   Ref0 h = new Ref0();
 2:   Object[] a = m1(mc);
 3:   Object[] e = m2(2*mc,h);
    }
    Object[] m1(int k) {
 4:   int i;
 5:   Ref0 l = new Ref0();
 6:   Object[] b = newA Object[k];
 7:   for(i=1;i<=k;i++)  {
 8:       b[i-1] = m2(i,l);
      }
 9:   Object[] c = newA Integer[9];
10:   return b;
    }

class Ref0 {
    Object ref;
}
```

```
    Object[] m2(int n, Ref0 s) {
11:     int j;
12:     Object c,d,e;
13:     Object[] f = newA Object[n]

14:     for(j=1;j<=n;j++) {
15:        if(j % 3 == 0) {
16:            c = newA Integer[j*2+1];
        }
        else {
17:            c = new Integer(0);
        }
18:        d = newA Integer[4];
19:        f[j-1] = c;
        }
20:     e = newA Integer[1];
21:     s.ref = e;
22:     return f;
    }
```

Figura 5.14: Programa ejemplo tomado de [BGY05].

Este programa resulta interesante ya que alguno de sus métodos contiene ciclos, hay varias invocaciones a métodos y se genera un ciclo anidado, además se declaran y acceden *arrays* y se utilizan expresiones linealizables como las que incluyen el operador % (módulo). También hay descubrimiento de posiciones de memoria y referencias como valor de retorno de los métodos.

La versión *Jimple* del programa se muestra a continuación

```
public class input.EjemploCC extends java.lang.Object {

    void m0(int)
    {
        input.EjemploCC this;
        int mc, $i0;
        input.Ref0 $r0, h;
        java.lang.Object[] a, e;

        this := @this: input.EjemploCC;
        mc := @parameter0: int;
        $r0 = new input.Ref0;
        specialinvoke $r0.<input.Ref0: void <init>()>();
        h = $r0;
        a = virtualinvoke this.<input.EjemploCC: java.lang.Object[] m1(int)>(mc);
        $i0 = 2 * mc;
        e = virtualinvoke this.<input.EjemploCC: java.lang.Object[] m2(int,input.Ref0)>($i0, h);
        return;
    }

    java.lang.Object[] m1(int)
    {
        input.EjemploCC this;
        int k, i, $i0;
        input.Ref0 $r0, l;
        java.lang.Object[] b, $r1;
        java.lang.Integer[] c;

        this := @this: input.EjemploCC;
        k := @parameter0: int;
        $r0 = new input.Ref0;
        specialinvoke $r0.<input.Ref0: void <init>()>();
        l = $r0;
        b = newarray (java.lang.Object)[k];
        i = 1;
        goto label1;

     label0:
        $i0 = i - 1;
        $r1 = virtualinvoke this.<input.EjemploCC: java.lang.Object[] m2(int,input.Ref0)>(i, l);
        b[$i0] = $r1;
        i = i + 1;

     label1:
        if i <= k goto label0;

        c = newarray (java.lang.Integer)[9];
        return b;
    }

    java.lang.Object[] m2(int, input.Ref0)
    {
        input.EjemploCC this;
        int n, j, $i0, $i1;
        input.Ref0 s;
        java.lang.Object[] f;
        java.lang.Object c;
        java.lang.Integer[] d, e;
        java.lang.Integer $r0;
```

```
        this := @this: input.EjemploCC;
        n := @parameter0: int;
        s := @parameter1: input.Ref0;
        f = newarray (java.lang.Object)[n];
        j = 1;
        goto label3;

    label0:
        $i0 = j % 3;
        if $i0 != 0 goto label1;

        $i0 = j * 2;
        $i0 = $i0 + 1;
        c = newarray (java.lang.Integer)[$i0];
        goto label2;

    label1:
        $r0 = new java.lang.Integer;
        specialinvoke $r0.<java.lang.Integer: void <init>(int)>(0);
        c = $r0;

    label2:
        d = newarray (java.lang.Integer)[4];
        $i1 = j - 1;
        f[$i1] = c;
        j = j + 1;

    label3:
        if j <= n goto label0;

        e = newarray (java.lang.Integer)[1];
        s.<input.Ref0: java.lang.Object ref> = e;
        return f;
    }
}
```

Las restricciones obtenidas en los puntos del programa[3] que han sido considerados interesantes son las siguientes.

- Para el punto del programa $3^{después}$ la restricción obtenida es[4]:

  $\{mc = 0 \wedge ar_6^{Object}.lth = 0 \wedge \$i0 = 0 \wedge ar_{13'}^{Object}.lth = 0 \wedge ar_{20}^{Integer}.lth = 1\} \cup$

  $\{2 * mc = ar_{13'}^{Object}.lth \wedge 2 * ar_6^{Object}.lth = ar_{13'}^{Object}.lth \wedge \$i0 = ar_{13'}^{Object}.lth \wedge$
  $ar_{20}^{Integer}.lth = 1\}$

  la abstracción del *heap* es:

[3] Los puntos del programa se han marcado en la sintaxis concreta (figura 5.14) ya que de esta forma se facilita la interpretación de las restricciones obtenidas

[4] donde lth es la abreviatura de length.

$$
\begin{array}{lll}
\{ & cs_1^{Ref0} & \rightarrow \{\ h\ \}, \\
& ar_6^{Object} & \rightarrow \{\ a\ \}, \\
& ar_{13'}^{Object} & \rightarrow \{\ e\ \}, \\
& ar_{20}^{Integer} & \rightarrow \{\ cs_1^{Ref0}.ref\ \}, \\
& ar_{13}^{Object} & \rightarrow \{\ ar_6^{Object}[?]\ \}, \\
& ar_{16}^{Integer} & \rightarrow \{\ ar_{13}^{Object}[?]\ \}, \\
& cs_{17}^{Integer} & \rightarrow \{\ ar_{13}^{Object}[?]\ \}, \\
& ar_{16'}^{Integer} & \rightarrow \{\ ar_{13'}^{Object}[?]\ \}, \\
& cs_{17'}^{Integer} & \rightarrow \{\ ar_{13'}^{Object}[?]\ \}, \\
\}
\end{array}
$$

$\{\ ar_{13'}^{Object}[?], ar_6^{Object}[?], ar_{13}^{Object}[?]\ \}$

$\{\ ar_{16}^{Integer}, cs_{17}^{Integer}, ar_{13}^{Object}, ar_{16'}^{Integer}, ar_{16'}^{Integer}\ \}$

Algunas aclaraciones pertinentes son las siguientes. Debido que que todas las posiciones de un *array* se abstraen en una única posición, se utiliza el símbolo " ? " para denotar el índice de esa única posición. Por ejemplo, $ar_6^{Object}[?]$ denota la única posición del *array* de objetos instanciado en la línea 6 del programa. Por otro lado, la variable $\$i0$, introducida automáticamente en la conversión a *Jimple*, corresponde a la expresión $2 * mc$. Finalmente, para algunas posiciones de memoria existen dos versiones diferentes, una que lleva el símbolo " $'$ " (prima) y otra que no. Por ejemplo, esta situación se presenta en las posiciones $ar_{13}^{Object}$ y $ar_{13'}^{Object}$. Esto se debe a que cada posición de memoria corresponde a una cadena de invocaciones diferente, y es posible distinguirlas. La posición $ar_{13}^{Object}$ es la que se instancia cuando el método $m0$ invoca al $m1$ y éste al $m2$. La posición $ar_{13'}^{Object}$ corresponde a cuando $m0$ invoca a $m2$. Distinguir las posiciones de memoria de acuerdo a las cadenas de invocación permite incrementar la precisión del análisis.

- Para el punto del programa $8^{antes}$ la restricción obtenida es:

  $\{k = ar_6^{Object}.lth \wedge i = 1 \wedge k \geq i\} \cup \{k = ar_6^{Object}.lth \wedge \$i0 \geq 0 \wedge i \geq 2 \wedge k \geq i\ \}$

  la abstracción del *heap* es:

$$
\begin{array}{lll}
\{ & cs_5^{Ref0} & \rightarrow \{\ l\ \}, \\
& ar_6^{Object} & \rightarrow \{\ b\ \}, \\
& ar_{13}^{Object} & \rightarrow \{\ ar_6^{Object}[?]\ \}, \\
& ar_{16}^{Integer} & \rightarrow \{\ ar_{13}^{Object}[?]\ \}, \\
& cs_{17}^{Integer} & \rightarrow \{\ ar_{13}^{Object}[?]\ \}, \\
& ar_{20}^{Integer} & \rightarrow \{\ cs_5^{Ref0}.ref\ \}, \\
\}
\end{array}
$$

  $\{\ ar_6^{Object}[?], ar_{13}^{Object}[?]\ \}$

  $\{\ ar_{13}^{Object}, ar_{16}^{Integer}, cs_{17}^{Integer}, ar_{20}^{Integer}\ \}$

  La variable $\$i0$, introducida automáticamente en la conversión a *Jimple*, corresponde a la expresión $i - 1$

- Para el punto del programa $8^{después}$ la restricción obtenida es:

  $\{k = ar_6^{Object}.lth \wedge i \geq 1 \wedge \wedge \$i0 \geq 0 \wedge k \geq 1\ \}$

  la abstracción del *heap* es la misma que para el punto $8^{antes}$.

Como puede observarse luego de la invocación de $m2$ la relación entre las variables $i$ y $k$ se pierde. Esta pérdida se debe al operador de *widening* y evitarla constituye uno de los aspectos a considerar en el trabajo a futuro.

- Para el punto del programa $9^{después}$ la restricción obtenida es:

  $\{k = 0 \wedge ar_6^{Object}.lth = 0 \wedge i = 1 \wedge ar_9^{Integer}.lth = 9\} \cup$

  $\{k = ar_6^{Object}.lth \wedge k \geq 0 \wedge ar_9^{Integer}.lth = 9 \wedge \$i0 \geq 1 \wedge i \geq k + 1\ \}$

  la abstracción del *heap* es:

  $$\begin{array}{ll}\{ & cs_5^{Ref0} \rightarrow \{\ l\ \},\\ & ar_6^{Object} \rightarrow \{\ b\ \},\\ & ar_{13}^{Object} \rightarrow \{\ ar_6^{Object}[?]\ \},\\ & ar_{16}^{Integer} \rightarrow \{\ ar_{13}^{Object}[?]\ \},\\ & cs_{17}^{Integer} \rightarrow \{\ ar_{13}^{Object}[?]\ \},\\ & ar_{20}^{Integer} \rightarrow \{\ cs_5^{Ref0}.ref\ \},\\ & ar_9^{Integer} \rightarrow \{\ c\ \},\\ \}\\ \{\ ar_6^{Object}[?], ar_{13}^{Object}[?]\ \}\\ \{\ ar_{13}^{Object}, ar_{16}^{Integer}, cs_{17}^{Integer}, ar_{20}^{Integer}\ \}\end{array}$$

- Para el punto del programa $15^{antes}$ la restricción obtenida es:

  $\{n = ar_{13}^{Object}.lth \wedge j = 1 \wedge n \geq j\} \cup$

  $\{n = ar_{13}^{Object}.lth \wedge \$i1 = j - 2 \wedge j \geq 2 \wedge n \geq j\ \}$

  la abstracción del *heap* es:

  $$\begin{array}{ll}\{ & sa_{null}^{Ref0} \rightarrow \{\ s\ \},\\ & ar_{13}^{Object} \rightarrow \{\ f\ \},\\ & ar_{16}^{Integer} \rightarrow \{\ c, ar_{13}^{Object}[?]\ \},\\ & cs_{17}^{Integer} \rightarrow \{\ c, ar_{13}^{Object}[?]\ \},\\ & ar_{18}^{Integer} \rightarrow \{\ d\ \},\\ \}\\ \{\ s, ar_{13}^{Object}[?]\ \}\\ \{\ ar_{16}^{Integer}, cs_{17}^{Integer}, ar_{18}^{Integer}\ \}\end{array}$$

- Para el punto del programa $16^{antes}$ la restricción obtenida es:

  $\{n = ar_{13}^{Object}.lth \wedge j\ \%\ 3 = 0 \wedge n \geq j \wedge j \geq 1\}$

  donde la relación $j\ \%\ 3 = 0$ en realidad es representada por la introducción de una variable temporal $tmp$ y la expresión $j = 3 * tmp$.

  la abstracción del *heap* es la misma que la obtenida para el punto del programa $15^{antes}$.

- Para el punto del programa $17^{antes}$ la restricción obtenida es:

  $\{n = ar_{13}^{Object}.lth \wedge n \geq j \wedge j \geq 1 \wedge j = 3 * tmp + \$i0 \wedge 1 \leq \$i0 \leq 2\} \cup$

  $\{n = ar_{13}^{Object}.lth \wedge n \geq j \wedge j \geq 1 \wedge j = 3 * tmp + \$i0 \wedge \$i0 \leq -1\}$

la abstracción del *heap* es la misma que la obtenida para el punto del programa $15^{antes}$.

- Para el punto del programa $21^{después}$ la restricción obtenida es:

  $\{n = O \wedge ar_{13}^{Object}.lth = 0 \wedge j = 1 \wedge ar_{20}^{Integer}.lth = 1\} \cup$

  $\{j = n + 1 \wedge ar_{13}^{Object}.lth = n \wedge ar_{20}^{Integer}.lth = 1\}$

  la abstracción del *heap* es:

  $$\begin{array}{lll}
  \{ & sa_{null}^{Ref0} & \rightarrow \{\ s\ \}, \\
  & ar_{13}^{Object} & \rightarrow \{\ f\ \}, \\
  & ar_{16}^{Integer} & \rightarrow \{\ c, ar_{13}^{Object}[?]\ \}, \\
  & cs_{17}^{Integer} & \rightarrow \{\ c, ar_{13}^{Object}[?]\ \}, \\
  & ar_{18}^{Integer} & \rightarrow \{\ d\ \}, \\
  & ar_{20}^{Integer} & \rightarrow \{\ e, sa_{null}^{Ref0}.ref\ \}, \\
  \}
  \end{array}$$

  $\{\ s, ar_{13}^{Object}[?]\ \}$

  $\{\ ar_{16}^{Integer}, cs_{17}^{Integer}, ar_{18}^{Integer}\ \}$

# Capítulo 6

# Trabajo a futuro y conclusiones

## 6.1. Trabajo a futuro

En [App98] se enuncia y demuestra el *teorema del pleno empleo para programadores de compiladores*. Este teorema demuestra que para cada compilador que realice optimización de código, siempre será posible programar uno nuevo que realice mejores optimizaciones. A partir de estas ideas es posible enunciar el siguiente teorema.

**Teorema 6.1.1** *Para cada programa $P$ que realiza descubrimiento automático de restricciones lineales entre variables de un programa, siempre existe otro programa $P'$ que encuentra restricciones más precisas. Donde una restricción $\alpha$ es más precisa que otra restricción $\beta$ si y solo si $\alpha \subset \beta$.*

**Demostración:** Se parte de un programa $P$ que realiza descubrimiento automático de restricciones lineales entre variables de un programa. Si se considera que a partir del resultado que se obtiene de $P$ se puede aproximar la post-condición del método que se analiza (ver sección 1.2.3 y definición 3.3.11), entonces puede asegurarse que existe un método $M_1$ cuya post-condición es { `FALSE` }, o sea el método no termina, pero la post-condición $\gamma$ que se obtiene del análisis $P(M_1)$ es tal que { `FALSE` } $\neq \gamma$. Si no existiera tal método el programa $P$ podría ser utilizado para resolver el *halting problem* que se conoce no computable. Es por esto que se puede considerar que el método $M_1$ efectivamente existe y es un método que al ser ejecutado no termina, pero el análisis que se obtiene a partir de $P$ no lo logra detectar.

Luego se construye un nuevo programa $P'$ de la siguiente manera. $P'(M) = P(M)$ siempre y cuando $M \neq M_1$, y $P'(M_1)$ da como resultado las mismas restricciones que $P(M_1)$ salvo para las sentencias que sean retornos del método en cuyo caso la restricción correspondiente es { `FALSE` }. Ahora el programa $P'$ sí descubre que el método $M_1$ no termina, pero tampoco puede ser utilizado para resolver el *halting problem*. Entonces debe existir un método $M_2$ cuya post-condición sea { `FALSE` } pero la post-condición $\delta$ que se obtiene del análisis $P'(M_2)$ es tal que { `FALSE` } $\neq \delta$. Análogamente se puede construir un nuevo programa $P''$ que tampoco puede ser utilizado para resolver el *halting problem*, y así sucesivamente se puede construir una sucesión infinita de programas cuya precisión va en aumento. □

El teorema 6.1.1 puede declararse como el *teorema del pleno empleo para programadores de analizadores estáticos que descubren restricciones lineales entre variables.* Dicho teorema sirve para asegurar que, siempre que se quiera, será posible aumentar la precisión del analizador, o dicho de otra forma, el trabajo a futuro es infinito.

Sin embargo no es razonable que la precisión del análisis sea incrementada de la forma presentada en la demostración del teorema, sino que lo lógico es trabajar en otros aspectos del programa.

Algunas de las posibilidades de trabajo a futuro son:

- Realizar la formalización de las ideas presentadas en el capítulo 5 (ver sección 5.6). Para esto es necesario demostrar que la abstracción del *heap* es un reticulado y luego que la función generada por las ecuaciones de *Data Flow* es monótona.

- Definir el análisis interprocedural simbólico para el lenguaje del capítulo 5. En este sentido gran parte del trabajo ya está hecho gracias a a incorporación del *shape analysis* que se realiza cuando se analiza un método con parámetros que son referencias a objetos. La parte restante sería hacer coincidir los argumentos concretos con la abstracción de los parámetros que se realiza al analizar el método invocado.

- Mejorar la técnica de *widening*. El *widening* es una de las principales fuentes de pérdida de precisión al realizar el análisis, por lo cual resultaría muy importante pulirlo para incrementar la precisión de las restricciones obtenidas. En [BHZ04], además del *widening* utilizado en este trabajo, se presenta otra técnica basada en certificados de convergencia finita. Resultaría interesante la implementación de esta última técnica y la comparación con la utilizada en este trabajo. Además es posible realizar algún tipo de técnica ad hoc en donde el *widening* se basa en alguna técnica estándar, pero se combina con alguna técnica adicional creada especialmente para el caso. Por ejemplo, muchas veces el *widening* hace perder la relación entre el índice que se incrementa en un ciclo y su cota, como es el caso de la restricción en el punto $8^{despues}$ de la figura 4.2. Una alternativa de mejora de la precisión es detectar que ni la cota ni el índice han sido modificados utilizando algún análisis previo y luego forzar la relación determinada por la guarda en los lugares en los que el *widening* la pierda.

- Migrar la implementación para utilizar la *Parma Polyhedra Library* (*PPL*) [BRZH02]. Esta migración tendría por, un lado, el beneficio de que los operadores de *widening* mencionados anteriormente ya están implementados como operadores nativos dentro de la librería. Además parecería ser una librería mucho más robusta que *PolyLib* y permitiría que la aplicación escale. Cabe destacar que debido a que el reticulado se comporta como un plug-in dentro del sistema, la propia arquitectura permitiría que la migración sea relativamente sencilla.

- Extender la técnica para permitir invocaciones recursivas. Esto no ha sido estudiado en profundidad, pero al menos existiría la posibilidad de tomar las ideas de [CC02] para considerar el análisis de una invocación recursiva como un *worst-case separate analysis*. De esta forma se considera que no se sabe nada del método invocado y se actúa de forma conservadora perdiendo toda la información que corresponda. Otra posibilidad es, al realizar análisis interprocedural mediante *inlining*, definir

una heurística mediante la cual se define la profundidad máxima de la cadena de invocaciones recursivas y la siguiente invocación sería considerada mediante *worst-case separate analysis*. Para el caso del análisis interprocedural simbólico, las invocaciones recursivas podrían analizarse mediante la utilización punto fijo.

- Incorporar nuevos tipos de pasaje de parámetros a la técnica, como por ejemplo, pasaje por referencia. En algunos lenguajes existe la posibilidad de pasar argumentos por referencia y resultaría necesario extender la técnica para poder contemplar dicha situación.

## 6.2. Conclusiones

La principal conclusión que puede obtenerse luego de haber definido e implementado las tres técnicas presentadas en el capítulo 3 de este trabajo es que, a pesar de haber demostrado que la obtención de restricciones de precisión absoluta no es computable, sí es posible obtener automáticamente restricciones de una precisión relativamente aceptable, pudiendo en varios casos obtener automáticamente restricciones de precisión absoluta. Por otro lado, gran parte de la pérdida de precisión presente en las restricciones descubiertas se debe fundamentalmente a la utilización del operador de *widening*, necesario para asegurar la terminación del análisis estático.

A pesar de que las ideas del capítulo 5 no han sido formalizadas, los resultados obtenidos con la herramienta que las implementa parecen alentadores. Una característica destacable de estas ideas es que combinan varias técnicas ya existentes de forma tal de lograr su objetivo; logrando, en algunos casos, que dicha combinación presente fenómenos muy interesantes como los que se describen en la sección 5.5.

A nuestro entender, la herramienta implementada es la única de sus características que permite trabajar con un subconjunto significativo de un lenguaje de programación real y moderno como es Java. Si bien aun posee algunos inconvenientes relacionados principalmente a la escalabilidad.

Como puede verse en la sección 6.1, existe mucho trabajo a futuro a realizar. Sin embargo, esta tesis representa un interesante punto de partida para el descubrimiento automático mediante análisis estático de restricciones lineales entre variables de programas para lenguajes con características de orientación a objetos.

# Apéndice A

# Demostraciones

**Proposición A.0.1** *Sea $F$ la función:*

$$F(\overrightarrow{RL}) = (F_1(\overrightarrow{RL}), F_2(\overrightarrow{RL}), F_3(\overrightarrow{RL}), ..., F_{2n}(\overrightarrow{RL}))$$

*obtenida a partir de un sistema de ecuaciones de* Data Flow *con $2n$ ecuaciones y $2n$ incógnitas, entonces:*

$$\overrightarrow{RL} \sqsubseteq \overrightarrow{RL}' \Rightarrow F(\overrightarrow{RL}) \sqsubseteq F(\overrightarrow{RL}')$$

*o sea:*

$$\overrightarrow{RL} \sqsubseteq \overrightarrow{RL}' \Rightarrow (\forall i : 1 \leq i \leq 2n \Rightarrow F_i(\overrightarrow{RL}) \subseteq F_i(\overrightarrow{RL}'))$$

**Demostración:** Cada una de las $F_i$ tiene alguna de las siguientes formas:

1. si $F_i$ es de la forma $F_i(x) = D^{\alpha}_{universal}$ y $\overrightarrow{RL} \sqsubseteq \overrightarrow{RL}'$, entonces:
   $F_i(\overrightarrow{RL}) = D^{\alpha}_{universal}$
   $F_i(\overrightarrow{RL}') = D^{\alpha}_{universal}$ y
   $D^{\alpha}_{universal} \subseteq D^{\alpha}_{universal}$ y en conclusión:
   $F_i(\overrightarrow{RL}) \subseteq F_i(\overrightarrow{RL}')$

2. si $F_i$ es de la forma

   $$F_i(x) = \bigcup_{\delta(j)} x_j$$

   y
   $\overrightarrow{RL} \sqsubseteq \overrightarrow{RL}'$, entonces:
   $F_i(\overrightarrow{RL}) = \overrightarrow{RL}_{\alpha} \cup \overrightarrow{RL}_{\beta} \cup ... \cup \overrightarrow{RL}_{\kappa}$
   $F_i(\overrightarrow{RL}') = \overrightarrow{RL}'_{\alpha} \cup \overrightarrow{RL}'_{\beta} \cup ... \cup \overrightarrow{RL}'_{\kappa}$

$\overrightarrow{RL}_\alpha \cup \overrightarrow{RL}_\beta \cup ... \cup \overrightarrow{RL}_\kappa \subseteq \overrightarrow{RL}'_\alpha \cup \overrightarrow{RL}'_\beta \cup ... \cup \overrightarrow{RL}'_\kappa$ por lema 1 y asociatividad de unión de conjuntos. En conclusión

$F_i(\overrightarrow{RL}) \subseteq F_i(\overrightarrow{RL}')$

3. si $F_i$ es de la forma $F_i(x) = x_j$ y $\overrightarrow{RL} \sqsubseteq \overrightarrow{RL}'$, entonces:

   $F_i(\overrightarrow{RL}) = \overrightarrow{RL}_j$

   $F_i(\overrightarrow{RL}') = \overrightarrow{RL}'_j$ y

   $\overrightarrow{RL}_j \subseteq \overrightarrow{RL}'_j$ por hipótesis. En conclusión:

   $F_i(\overrightarrow{RL}) \subseteq F_i(\overrightarrow{RL}')$

4. si $F_i$ es de la forma $F_i(x) = x_j \cap D_\alpha$ y $\overrightarrow{RL} \sqsubseteq \overrightarrow{RL}'$, entonces:

   $F_i(\overrightarrow{RL}) = \overrightarrow{RL}_j \cap D_\alpha$

   $F_i(\overrightarrow{RL}') = \overrightarrow{RL}'_j \cap D_\alpha$ y

   $\overrightarrow{RL}_j \cap D_\alpha \subseteq \overrightarrow{RL}'_j \cap D_\alpha$ por lema 2. En conclusión:

   $F_i(\overrightarrow{RL}) \subseteq F_i(\overrightarrow{RL}')$

5. si $F_i$ es de la forma $F_i(x) = ((((x_j \nearrow 1) \cap D_\alpha) \parallel k) \cap D_\beta) \searrow dimension(x_j) + 1$ y $\overrightarrow{RL} \sqsubseteq \overrightarrow{RL}'$, entonces:

   $F_i(\overrightarrow{RL}) = ((((\overrightarrow{RL}_j \nearrow 1) \cap D_\alpha) \parallel k) \cap D_\beta) \searrow dimension(x_j) + 1$

   $F_i(\overrightarrow{RL}') = ((((\overrightarrow{RL}'_j \nearrow 1) \cap D_\alpha) \parallel k) \cap D_\beta) \searrow dimension(x_j) + 1$

   y puede realizarse el siguiente desarrollo:

   $(\overrightarrow{RL}_j \nearrow 1) \subseteq (\overrightarrow{RL}'_j \nearrow 1)$ por lema 3

   $((\overrightarrow{RL}_j \nearrow 1) \cap D_\alpha) \subseteq ((\overrightarrow{RL}'_j \nearrow 1) \cap D_\alpha)$ por lema 2

   $(((\overrightarrow{RL}_j \nearrow 1) \cap D_\alpha) \parallel k) \subseteq (((\overrightarrow{RL}'_j \nearrow 1) \cap D_\alpha) \parallel k)$ por lema 5

   $((((\overrightarrow{RL}_j \nearrow 1) \cap D_\alpha) \parallel k) \cap D_\beta) \subseteq ((((\overrightarrow{RL}'_j \nearrow 1) \cap D_\alpha) \parallel k) \cap D_\beta)$ por lema 2

   y finalmente:

   $((((\overrightarrow{RL}_j \nearrow 1) \cap D_\alpha) \parallel k) \cap D_\beta) \searrow dimension(x_j) + 1 \subseteq$

   $((((\overrightarrow{RL}'_j \nearrow 1) \cap D_\alpha) \parallel k) \cap D_\beta) \searrow dimension(x_j) + 1$ por lema 4

   En conclusión:

   $F_i(\overrightarrow{RL}) \subseteq F_i(\overrightarrow{RL}')$

6. si $F_i$ es de la forma $F_i(x) = x_j \parallel k$ y $\overrightarrow{RL} \sqsubseteq \overrightarrow{RL}'$, entonces:

   $F_i(\overrightarrow{RL}) = \overrightarrow{RL}_j \parallel k$

   $F_i(\overrightarrow{RL}') = \overrightarrow{RL}'_j \parallel k$ y

   $\overrightarrow{RL}_j \parallel k \subseteq \overrightarrow{RL}'_j \parallel k$ por lema 5. En conclusión:

   $F_i(\overrightarrow{RL}) \subseteq F_i(\overrightarrow{RL}')$

□

**Lema 1** *Sean $A$, $B$, $A'$ y $B'$ cuatro conjuntos que cumplen la siguiente hipótesis: $A \subseteq A' \wedge B \subseteq B'$, entonces se cumple que $A \cup B \subseteq A' \cup B'$.*

**Demostración:** Se supone que $A \cup B \nsubseteq A' \cup B'$, entonces debe existir algún elemento $e$ tal que: $e \in A \cup B \wedge e \notin A' \cup B'$. Como $e \in A \cup B$, entonces $e \in A \vee e \in B$. Se asume que $e \in A$ entones $e \in A'$ por hipótesis y en consecuencia se cumple que $e \in A' \cup B'$ y se alcanza un absurdo. Si se asume que $e \in B$ se alcanza análogamente un absurdo. En conclusión no existe el elemento $e$ y $A \cup B \subseteq A' \cup B'$. $\square$

**Lema 2** *Sean $A$, $A'$ y $B$ tres conjuntos que cumplen la siguiente hipótesis: $A \subseteq A'$, entonces se cumple que $A \cap B \subseteq A' \cap B$.*

**Demostración:** Se supone que $A \cap B \nsubseteq A' \cap B$, entonces debe existir algún elemento $e$ tal que: $e \in A \cap B \wedge e \notin A' \cap B$. Como $e \in A \cap B$, entonces $e \in A \wedge e \in B$. Entonces $e \in A' \wedge e \in B$ por hipótesis y en consecuencia se cumple que $e \in A' \cap B$ y se alcanza un absurdo. En conclusión no existe el elemento $e$ y $A \cap B \subseteq A' \cap B$. $\square$

**Lema 3** *Sean $D_1$ y $D_2$ dos dominios de poliedros convexos cerrados de dimensión $k$ que cumplen la siguiente hipótesis: $D_1 \subseteq D_2$, entonces se cumple que $D_1 \nearrow 1 \subseteq D_2 \nearrow 1$.*

**Demostración:** Se supone que $D_1 \nearrow 1 \nsubseteq D_2 \nearrow 1$, entonces debe existir algún elemento $e = (e_1, e_2, ..., e_k, e_{k+1})$ tal que: $e \in D_1 \nearrow 1 \wedge e \notin D_2 \nearrow 1$. Como $e \in D_1 \nearrow 1$, entonces $(e_1, e_2, ..., e_k) \in D_1$, entonces $(e_1, e_2, ..., e_k) \in D_2$ por hipótesis. Pero si $(e_1, e_2, ..., e_k) \in D_2$, entonces $(e_1, e_2, ..., e_k, z) \in D_2 \nearrow 1 \ \forall z : z \in \mathbf{Z}$ por definición del operador. En particular $z = e_{k+1}$, entonces $e = (e_1, e_2, ..., e_k, e_{k+1}) \in D_2 \nearrow 1$ y se llega a un absurdo por suponer que $D_1 \nearrow 1 \nsubseteq D_2 \nearrow 1$ $\square$

**Lema 4** *Sean $D_1$ y $D_2$ dos dominios de poliedros convexos cerrados de dimensión $k$ que cumplen la siguiente hipótesis: $D_1 \subseteq D_2$, entonces, si $1 \leq i \leq k$, se cumple que $D_1 \searrow i \subseteq D_2 \searrow i$.*

**Demostración:** Se supone que $D_1 \searrow i \nsubseteq D_2 \searrow i$, entonces debe existir algún elemento $e = (e_1, ..., e_{i-1}, e_{i+1}, ..., e_k)$ tal que: $e \in D_1 \searrow i \wedge e \notin D_2 \searrow i$. Como $e \in D_1 \searrow i$, entonces $\exists j : (e_1, ..., e_{i-1}, j, e_{i+1}, ..., e_k) \in D_1$ por definición del operador, entonces $(e_1, ..., e_{i-1}, j, e_{i+1}, ..., e_k) \in D_2$ por hipótesis. Luego $e = (e_1, ..., e_{i-1}, e_{i+1}, ..., e_k) \in D_2 \searrow i$ nuevamente por definición del operador, y se llega a un absurdo por suponer que $D_1 \searrow i \nsubseteq D_2 \searrow i$ $\square$

**Lema 5** *Sean $D_1$ y $D_2$ dos dominios de poliedros convexos cerrados de dimensión $k$ que cumplen la siguiente hipótesis: $D_1 \subseteq D_2$, entonces, si $1 \leq i \leq k$, se cumple que $D_1 \parallel i \subseteq D_2 \parallel i$.*

**Demostración:** Se supone que $D_1 \parallel i \not\subseteq D_2 \parallel i$, entonces debe existir algún elemento $e = (e_1, ..., e_{i-1}, e_i, e_{i+1}, ..., e_k)$ tal que: $e \in D_1 \parallel i \wedge e \notin D_2 \parallel i$. Si $e \in D_1 \parallel i$ entonces $\exists j$ : $(e_1, ..., e_{i-1}, j, e_{i+1}, ..., e_k) \in D_1$ por definición del operador, luego $(e_1, ..., e_{i-1}, j, e_{i+1}, ..., e_k) \in D_2$ por hipótesis. Pero entonces $(e_1, ..., e_{i-1}, z, e_{i+1}, ..., e_k) \in D_2 \parallel i \ \forall z : z \in \mathbf{Z}$ por definición del operador. En particular $z = e_i$ y entonces $e = (e_1, ..., e_{i-1}, e_i, e_{i+1}, ..., e_k) \in D_2 \parallel i$ y se llega a un absurdo. $\square$

# Apéndice B

# Interfaces del mecanismo de *plug-ins*

En la interfaz $LatticeItf$ se especifican las operaciones básicas que se requieren de un reticulado.

```
public interface LatticeItf {

    public void intersection(LatticeItf pLattice2, LatticeItf pOutput);

    public void union(LatticeItf pLattice2, LatticeItf pOutput);

    public void widening(LatticeItf pLattice2, LatticeItf pOutput);
}
```

La interfaz $PolyhedraPowersetItf$ extiende a $LatticeItf$ para incorporar las operaciones propias de un dominio de poliedros convexos cerrados.

Resulta interesante observar que el método $addVariables(List)$, se corresponde con el operador "$\nearrow$" descripto en la sección 2.1.2, el método $eliminateVar(AnalysisVariable)$ se corresponde con el operador "$\searrow$" y el método $reset(AnalysisVariable)$ se corresponde con el operador "$\|$". Además el método $overwrite(AnalysisVariable, AnalysisVariable)$ es el que se utiliza para realizar el *binding* de una variable con otra, para luego eliminar a la primera, situación que se presenta en la regla correspondiente a las sentencias que realizan definiciones, donde se agrega una variable temporal para conservar la información de la variable que se define.

```
public interface PolyhedraPowersetItf extends LatticeItf {

    public List getVars();

    public void addConstraints(Collection pConstraints);
    public void addDisjointConstraints(List pConstraints);

    public boolean hasVariable(AnalysisVariable pVar);

    public void addVariables(List pVars);

    public void reset(AnalysisVariable av);
```

```
    public void eliminateVar(AnalysisVariable pLocal);

    //new takes the place of old. And old is removed
    public void overwrite(AnalysisVariable pNew, AnalysisVariable pOld);
}
```

La interfaz $PolyhedraPowersetExtensionItf$ es la que se utiliza para extender los métodos de $PolyhedraPowersetItf$ de forma tal que representen el par compuesto por el dominio de poliedros convexos cerrados y la abstracción del *heap* utilizada en el capítulo 5. Aquí también se presenta una correspondencia entre los métodos de la interfaz y los operadores correspondientes a la abstracción del *heap* definidos en la sección 5.2.

```
public interface PolyhedraPowersetExtensionItf
    extends PolyhedraPowersetItf
{

    public void addSymbolicArgumentCreationSite(
                           ParameterRef            pParamRef,
                           AbstractedReference     pDefinedVar,
                           AnalysisVariableManager pAnalysisVariableManager);

    public void addCreationSite(NewExpr                pNewExpr,
                                Stmt                   pRelatedStmt,
                                AbstractedReference    pReference,
                                AnalysisVariableManager pAnalysisVariableManager,
                                ConstraintFactoryItf   pConstraintFactory,
                                boolean                pMayAlias);

    public void addCreationSite(NewArrayExpr           pNewExpr,
                                Stmt                   pRelatedStmt,
                                AbstractedReference    pReference,
                                AnalysisVariableManager pAnalysisVariableManager,
                                ConstraintFactoryItf   pConstraintFactory,
                                boolean                pMayAlias)
        throws PredictableNullPointerException;


    public void addAliasing(AbstractedReference     pTarget,
                            AbstractedReference     pSource,
                            Stmt                    pRelatedStmt,
                            AnalysisVariableManager pAnalysisVariableManager,
                            ConstraintFactoryItf    pConstraintFactory,
                            boolean                 pMay);

    public void addThisRef(ThisRef thisRef, AbstractedReference pReference);

    public void addNullAlias(AbstractedReference target, boolean may);

    public void eliminateReference(AbstractedReference pReference);
}
```

# Bibliografía


[App98] A. W. Appel. *Modern compiler implementation in Java*. Cambridge University Press, New York, NY, USA, 1998.

[BGP99] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, 1999.

[BGY05] V. Braberman, D. Garbervetsky, and S. Yovine. Synthesizing parametric specifications of dynamic memory utilization in object-oriented programs. In *FTfJP'2005: 7th Workshop on Formal Techniques for Java-like Programs*, Glasgow, Scotland, July 26, 2005.

[BHRZ03] R. Bagnara, P. Hill, E. Ricci, and E. Za. Precise widening operators for convex polyhedra, 2003.

[BHZ04] R. Bagnara, P. Hill, and E. Za. Widening operators for powerset domains, 2004.

[BRZH02] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis: Proceedings of the 9th International Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 213–229, Madrid, Spain, 2002. Springer-Verlag, Berlin.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC02] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In R.N. Horspool, editor, *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, pages 159–178, Grenoble, France, April 6—14 2002. LNCS 2304, Springer, Berlin.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[CL05a] B. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, pages 147–163, 2005.

[CL05b] B. E. Chang and K. R. M. Leino. Inferring object invariants: Extended abstract. *Electr. Notes Theor. Comput. Sci.*, 131:63–74, 2005.

[Cla96] P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: applications to analyze and transform scientific programs. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*, pages 278–285, New York, NY, USA, 1996. ACM Press.

[ECGN00] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *International Conference on Software Engineering*, pages 449–458, 2000.

[FR99] G. Filé and F. Ranzato. The powerset operator on abstract interpretations. *Theoretical Computer Science*, 222(1-2):77–111, July 1999.

[Hal79] N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme*. Thèse de 3ème cycle d'informatique, Université scientifique et médicale de Grenoble, Grenoble, France, March 1979.

[Lho02] O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[Loe99] V. Loechner. *PolyLib*: A library for manipulating parameterized polyhedra. Available at http://icps.u-strasbg.fr/ loechner/polylib/, March 1999. Declares itself to be a continuation of [Wil93].

[Min00] A. Miné. Representation of two-variable difference or sum constraint set and application to automatic program analysis. Master's thesis, ENS-DI, Paris, France, 2000. http://www.di.ens.fr/ mine/publi/report-mine-dea.pdf.

[MRTT53] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games – Volume II*, number 28 in Annals of Mathematics Studies, pages 51–73. Princeton University Press, Princeton, New Jersey, 1953.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3–valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[SSM04] S.Sankaranarayanan, H. B. Sipma, and Z. Manna. Constraint-based linear-relations analysis. In R. Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2004.

[TS05] N. Tillmann and W. Schulte. Parameterized unit tests with unit meister. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on*

*Foundations of software engineering*, pages 241–244, New York, NY, USA, 2005. ACM Press.

[VRHS+99] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot- A java optimization framework. In *CASCON'99*, pages 125–135, 1999.

[Wil93] D. K. Wilde. A library for doing polyhedral operations. Master's thesis, Oregon State University, Corvallis, Oregon, December 1993. Also published as IRISA *Publication interne* 785, Rennes, France, 1993.