Tesis de Licenciatura en Ciencias de la Computación:

# Predicción paramétrica de requerimientos de memoria. Especificación modular.

Martín Rouaux

Libreta universitaria: 207/00

martinfr@gmail.com

**Director:**

Dr. Diego Garbervetsky

**Codirector:**

Dr. Sergio Yovine

**Resumen**

Las comunidades de sistemas de tiempo real y embebidos han demostrado, en los últimos años, un interés creciente por los lenguajes orientados a objetos tipo Java. La adopción de este tipo de lenguajes en el contexto de estos sistemas requiere la solución de al menos dos grandes problemas: la impredictibilidad temporal, dada por las interrupciones relacionadas con la administración de memoria dinámica (garbage collector) y la capacidad de determinar requerimientos de memoria para una aplicación dada.

El estudio cuantitativo de requerimientos de memoria es al día de hoy un problema desafiante. Existen diferentes técnicas para atacar este problema sin embargo pocas son orientadas a lenguajes imperativos. Y, en general, las que sí lo son tienen problemas de escalabilidad.

Una alternativa interesante para atacar el problema de escalabilidad es definir un algoritmo composicional para la inferencia de cotas paramétricas del consumo de memoria. Es decir, dado un método $m$, queremos obtener una cota superior de la cantidad de memoria requerida para su ejecución a partir del análisis local de $m$ (sin considerar las llamadas que $m$ realiza) y de la especificación de consumo de los métodos llamados por $m$.

Al analizar los objetos creados durante la ejecución de un método $m$ podemos distinguir entre los objetos creados por $m$ y los objetos creados por los métodos invocados por $m$. Esta partición permite inferir expresiones de consumo para un método en función de los requerimientos locales y la especificación de consumo de los métodos llamados.

Este enfoque requiere la implementación de un conjunto de operadores que permita inferir la especificación de consumo de un método en función de las especificaciones de consumo de los métodos llamados. Estos operadores fueron implementados exitosamente.

Esta tesis presenta una técnica de análisis composicional para el estudio cuantitativo de requerimientos de memoria en programas Java. A su vez, presenta el diseño e implementación de un prototipo que fue utilizado para evaluar este nuevo enfoque.

# Agradecimientos

A todos aquellos que por muchos años insistieron en que realice este trabajo para completar la carrera entre ellos Nacho, Gise, Carina, Germán, Matías, Hernán, los padres de Alejandro y particularmente a mi familia. A mis padres, que forman parte de este primer grupo, pero a su vez fueron un soporte económico importante durante los primeros años de estudio.

A Diego y Sergio, mis directores por todo el apoyo y dedicación que me han dado este último tiempo. Sobre todo a Diego, con quién he pasado muchas horas tratando de sacar este trabajo adelante y ha sabido motivarme a investigar incluso no estando yo muy convencido. Debo agradecer además el trato de amigo que me ha dado siempre.

En lo académico debo agradecer a Victor quién se mantuvo al tanto de nuestros avances y aportó ideas para el desarrollo de este trabajo.

A Alejandro, quién además de ser un amigo incondicional aportó al desarrollo de esta tesis con ideas y por sobre todo aguantando mis continuas desiluciones.

La Universidad de Buenos Aires merece una mención aparte, la posibilidad de estudiar en una universidad pública y gratuita de la categoría de la UBA es algo de lo que debemos estar orgullosos.

Por último y no por eso menos importante, quisiera agradecer especialmente a Romi que tuvo que aguantarme mientras yo dedicaba gran parte de mi tiempo a este trabajo. El cariño y apoyo incondicional que me brindó son factores importantísimos para que hoy esté cerrando esta etapa. A ella, mi más sincero agradecimiento.

# Índice general

# Capítulo 1

# Introducción

## 1.1. Motivación

Las comunidades de sistemas embebidos de tiempo real han demostrado, en los últimos años, un interés creciente por los lenguajes orientados a objetos tipo Java. La facilidad para la encapsulación de abstracciones y la comunicación mediante interfaces bien definidas que proveen estos lenguajes, el gran número de herramientas y bibliotecas disponibles en estas tecnologías son algunos de los motivos de este interés.

Sin embargo, la adopción de este clase de lenguajes para este tipo de sistemas requiere la solución de al menos dos grandes problemas: la impredictibilidad temporal, dada por las interrupciones relacionadas con la administración de memoria dinámica (garbage collector) y la capacidad de determinar requerimientos de memoria para una aplicación dada.

Existen numerosos trabajos que intentan tratar el primero de estos problemas [BCG04, Hen98, HIB+02, RF02, BG00, CR04, GNYZ04], mientras que calcular requerimientos de memoria sigue siendo un problema desafiante. Evaluar de forma cuantitativa los requerimientos de memoria es un problema inherentemente difícil, de hecho, obtener una cota superior finita sobre el consumo de memoria es indecidible [Ghe02].

En [BGY06, BFGY08] se plantea una técnica original para aproximar los requerimientos de memoria de un programa Java. Dado un método $m$ con parámetros $P_m$ en [BGY06] se obtiene una cota superior paramétrica (en término de $P_m$) de la cantidad de memoria reservada por $m$ mediante instrucciones *new*. Considerando la liberación de memoria en [BFGY08] se obtiene una cota superior paramétrica de la cantidad de memoria necesaria para ejecutar de manera segura m y todos los métodos a los que este invoca. Esta expresión puede ser vista como una precondición que establece que m requiere a lo sumo esa cantidad de memoria disponible antes de su ejecución.

Para calcular esa estimación se considera la liberación de memoria que puede

ocurrir durante la ejecución del método. Básicamente, se adopta un modelo de administración de memoria basado en regiones [GNYZ04], donde el tiempo de vida de las regiones es asociado al tiempo de vida de los métodos. Para obtener una expresión del consumo máximo de un método, se modela el consumo de las posibles configuraciones de pilas de regiones. Este modelo lleva a un conjunto de problemas de maximización de polinomios, que son resueltos mediante una técnica basada en Bases de Bernstein [CFGV09] que obtiene una solución paramétrica.

Ambas técnicas requieren del conocimiento de las distintas configuraciones del stack de llamadas del método bajo análisis. Es decir, el método requiere conocer su contexto de llamada, condición que en general no es deseable. Esto determina que el anális no sea modular. Las ventajas de la modularidad son muchas: reutilización de especificaciones, mejor escalabilidad, capacidad de analizar aplicaciones que llaman a métodos no analizables, mejor capacidad para la integración con otras técnicas (por ejemplo [SR05]), etc.

Un enfoque interesante para atacar el problema de modularidad es la incorporación de un modelo de especificaciones que describa los efectos que produce, sobre la memoria, la ejecución de un método. La especificación de un método puede ser inferida mendiante una técnica de análisis estático que considere la memoria localmente reservada por un método (sin considerar las llamadas que realiza) y los efectos de las invocaciones que el método realiza.

En esta tesis atacamos el problema de determinar requerimientos de memoria para programas Java. El esfuerzo está principalmente enfocado en el desarrollo de un algoritmo composicional que permita sintetizar un resumen por método que describa los efectos en el *heap*. Dado un método $m$, el resumen debe describir la cantidad de objetos requeridos para la ejecución de $m$. A su vez, debe especificar los objetos que exceden el tiempo de vida de $m$ permitiendo seguir su evolución en los métodos que invocan a $m$. Este resumen es utilizado nuevamente al inferir el resumen de los métodos que llaman a $m$, de aquí proviene la designación de composicional. Siguiendo este enfoque creemos que es posible obtener algunos de los beneficios que la modularidad implica.

## 1.2. Acerca de este trabajo

Tomando como base las técnicas presentadas por [BGY06, BFGY08], esta tesis presenta un análisis composicional para la inferencia de requerimientos de memoria. Un análisis composicional que infiera resumenes por método que describan el efecto de invocar al método permitiría fortalecer la técnica de análisis obteniéndose mejoras en los siguientes aspectos:

- Reutilización de especificaciones, ya sea calculadas por la misma herramienta o provista por el programador.

- Mayor escalabilidad.
- Capacidad de analizar programas con métodos no analizables. Estos casos podrían ser especificados por el programador o bien podría disponerse de una especificación previa.
- Mayor capacidad de integración a otras técnicas.

Esta técnica se logra a partir del análisis local de consumo de un método y de la especificación de consumo de los métodos llamados. A partir de esta información se infieren los requerimientos de memoria del método bajo análisis. Para esto se realiza una partición de la memoria requerida por un método, la cual está dada por los objetos que son creados por el método bajo análisis y por los objetos que son creados por los métodos invocados. A su vez estas categorías pueden particionarse entre los objetos que exceden el tiempo de vida del método creador y aquellos que no. De esta forma podemos ver el consumo de un método $m$ de parámetros $p$ como $M_m(p) = Temp_m(p) + Res_m(p)$, donde $Temp_m(p)$ son los objetos que no exceden el tiempo de vida de $m$ (temporal de $m$) y $Res_m(p)$ son los objetos que si exceden el tiempo de vida de $m$ (residual de $m$).

Clasificando el temporal y residual en función del método creador: $M_m(p) = TempLocal_m(p) + TempCall_m(p) + ResLocal_m(p) + ResCall_m(p)$ donde $TempLocal_m(p)$ corresponde a los objetos creados por $m$ y $TempCall_m(p)$ son objetos residuales producto de la invocación a métodos que no exceden el tiempo de vida de $m$. Este mismo concepto se aplica a $ResLocal_m(p)$ y $ResCall_m(p)$.

La partición del consumo de memoria nos permite analizar en primer instancia el consumo local de un método y luego a partir de las especificaciones de los métodos llamados sintetizar expresiones de consumo para el método bajo análisis.

Este nuevo enfoque requiere:

1. Un mecanismo que permita obtener la cantidad de objetos requeridos por el método sin tener en cuenta el efecto de invocar a otros métodos.
2. Poder especificar el consumo de memoria de un método no sólo de forma cuantitativa, sino también de manera cualitativa, para poder determinar el tiempo de vida del conjunto de objetos cuyo tamaño se cuantifica.
3. Un mecanismo de inferencia que permita, a partir del análisis del cuerpo de un método $m$ y el conjunto de especificaciones de los métodos llamados por $m$ inferir el *resumen* que describe los efectos en memoria producidos por la ejecución de $m$.

A lo largo de este trabajo abordaremos cada uno de los requerimientos enunciados proveyendo técnicas que permiten implementarlos. El requerimiento 1 es atacado

mediante la técnica presentada [BGY06]. Para determinar el tiempo de vida de un objeto utilizamos el análisis propuesto en [SYG05].

Respecto al requerimiento 2, es abordado utilizando los resultados de [SYG05] que permiten modelar el efecto de la invocación de un método en cuanto a las relaciones entre los objetos. Este modelo es extendido con expresiones de consumo que permiten cuantificar los conjuntos de objetos que exceden el tiempo de vida de un método.

El requerimiento 3 requiere inferir expresiones de consumo del método bajo análisis en función de la especficación de los método llamados. Para esto manipulamos expresiones simbólicamente en función de invariantes que restringen las variables locales de un método y ligan éstas con los parámetros de los métodos llamados.

La solución propuesta plantea un operador de maximización que permite maximizar una expresión de consumo en un determinado dominio. Este operador fue implementado a partir de los resultados de [BFGY08]. Además utiliza un operador que suma expresiones de consumo en función del invariante de una llamada, esto permite modelar la acumulación de objetos producto del residuo de invocaciones a métodos. Esto se resuelve mediante la suma de un polinomio evaluado en todos los valores enteros de un conjunto de las variables.

La implementación de estos operadores son resueltas mediante la integración de la biblioteca *libbarvinok* [Ver07] para conteo del número de puntos enteros en politopos paramétricos.

### 1.2.1. Contribuciones

Esta tesis realiza cuatro contribuciones. Primero, propone la especificación de un resumen por método que describe los efectos que este produce en el *heap*. Este resumen modela los requerimientos de memoria de un método y las relaciones entre objetos que exceden el tiempo de vida del método.

La segunda contribución es la utilización de la técnica para el cálculo de consumo de memoria presentada en [BFGY08] en conjunto con el análisis de escape para un esquema de administración de memoria basado en regiones presentado en [SYG05].

La tercera contribución es un mecanismo de inferencia que permite sintetizar el resumen de un método $m$ a partir del conjunto de resumenes que describen el comportamiento de los métodos llamados por $m$.

Finalmente, se presenta una evaluación de la técnica propuesta. Se define un caso de estudio en donde se infieren los resumenes de todos los métodos que componen un programa utilizando la herramienta implementada a lo largo de este trabajo.

## 1.3. Visión General

Como fue mencionado en la sección anterior, el objetivo principal de este trabajo es desarrollar una técnica de análisis composicional para el estudio de requerimientos

de memoria en programas Java. A su vez, se pretende desarrollar una herramienta que permita realizar una prueba conceptual sobre la técnica desarrollada.

Para el cálculo de consumo se asume un esquema de administración de memoria basado en *scopes* (ver sección 2.3.1). En este esquema, al finalizar la ejecución de un método $m$ solo pueden ser recolectados los objetos creados por $m$ o por un método al que $m$ invoca.

Existen diferentes esquemas de administración de memoria que soportan el manejo de regiones [Tab09, SYG05, BG00]. En [Tab09] se asocia el ciclo de vida de una región a un método. Los objetos son reservados en el espacio de la región que corresponda según su tiempo de vida. De esta manera se genera una estructura de pila de regiones con un comportamiento similar a la pila de llamadas a métodos.

Las regiones son creadas con un tamaño fijo cuyo valor puede ser especificado por una constante o una expresión paramétrica en función de los parámetros del método. Esta expresión es utilizada por el administrador de memoria para determinar, en tiempo de ejecución, el tamaño que debe asginar a una región al momento de su creación. Conocer el valor de estas expresiones permite la implementación de un algoritmo de administración de memoria eficiente.

Nuestro prototipo inicial permite sintetizar estimadores paramétricos que aproximan la cantidad total de objetos reservados (ver secciones 4.1.3 y 4.2) y la cantidad de objetos requeridos, considerando la liberación de memoria, por la ejecución de programas Java. La herramienta puede ser adaptada, sin mayor esfuerzo, para estimar la cantidad de memoria requerida si se considera el tamaño de los objetos creados.

Los componentes centrales de la solución propuesta son (ver figura 1.1) :

- Analizador de información de escape:
  - Análisis de escape: Aproxima de forma automática el tiempo de vida de los objetos(ver secciones 2.3.2 y 5.3).
  - Análisis interferencia de punteros: Aproxima de forma automática como se relacionan los objetos entre sí con el objeto de asignar objetos relacionados en la misma región (ver sección 5.3.1).
- Generador de invariantes locales: Es responsable de proveer los invariantes requeridos por las técnicas de predicción de memoria (ver sección 5.1).
- Calculadora de expresiones paramétricas: Resuelve las operaciones de conteo, maximización y suma requeridas por las técnicas de predicción de memoria (ver secciones 2.1.1 y 5.2).
- Analizador de memoria dinámica: Infiere estimadores paramétricos de la cantidad de objetos requeridos por un método (ver capítulo 4).

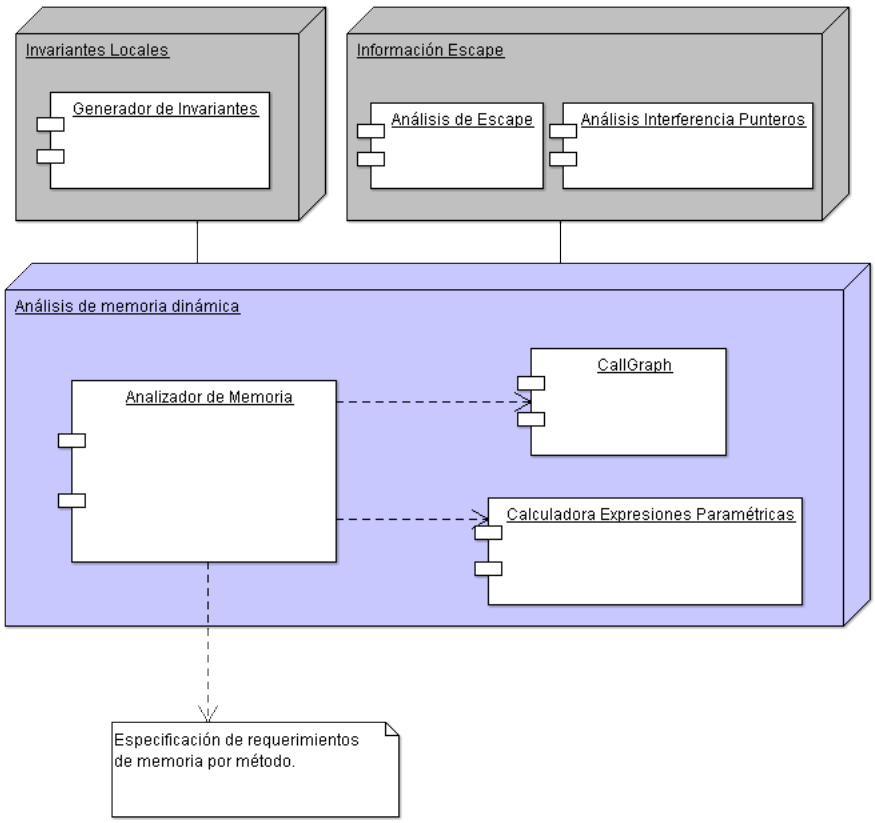


Figura 1.1: Componentes centrales que integran la solución propuesta.

La implementación del *Analizador de memoria dinámica* y la *Calculadora de expresiones paramétricas* constituyen el mayor esfuerzo de esta tesis.

El *Analizador de memoria dinámica*, a partir de los invariantes generados y la información obtenida por el análisis de escape, sintetiza expresiones paramétricas que acotan de manera conservadora la cantidad de objetos requeridos por la ejecución de un método.

La técnica desarrollada propone un algoritmo de inferencia de resumenes por método considerando el árbol de llamadas. Es decir, para obtener el resumen de un método $m$ se utiliza el conjunto de resumenes de los métodos llamados por $m$. Esto impone un orden sobre el conjunto de métodos a analizar, ya que si $m$ llama a $m'$ entonces debemos calcular primero el resumen de $m'$. Para esto, el algoritmo propuesto recorre el árbol de llamadas de forma bottom-up. Esto implica la necesidad de realizar operaciones de suma y maximización sobre los estimadores de consumo ya calculados. Estas operaciones son resueltas por la *Calculadora de expresiones paramétricas*.

Si bien, las operaciones de conteo y maximización son utilizadas en [BFGY08] e implementadas por [Ver07], al inicio de esta investigación no estaba claro como implementar el operador de suma. Por otro lado, la integración de [Ver07] a una herramienta Java presenta un desafío técnico no trivial.

El operador de suma, es resuelto por una técnica de conteo sobre un dominio paramétrico, asociando pesos a cada valuación que satisface el dominio [VB08]. [Ver07] implementa diferentes algoritmos para la resolución de este problema.

Los componentes que implementan el *Análisis de escape* y el *Análisis de interferencia de punteros* son resueltos mediante el análisis estático propuesto en [SYG05].

El *Generador de invariantes locales* es resuelto mediante la integración de la herramienta desarrollada por Garbervetsky en [Gar05].

## 1.4. Trabajos relacionados

La mayoría de los trabajos relacionados están enfocados en asegurar que los programas no violen políticas de recursos, las cuales son impuestas mediante el uso de sistemas de tipos enriquecidos [HJ03, CNQR05, HP99] o utilizando una lógica de programa [AM05, BHMS04, CEI+07, BPS05].

Por otra parte, hay pocos trabajos relacionados con la inferencia de requerimientos de memoria dinámica [Ghe02, HJ03, CJPS05, USL03, AAG+07, BGY06].

En [HJ03] se propone una solución para obtener cotas lineales sobre el uso de programas funcionales de primer orden. Un punto clave de la solución es el uso de sistemas de tipos lineales que permiten reciclar el espacio de la útlima estructura de datos utilizada. Con este enfoque es posible reciclar la memoria dentro de cada función pero no entre funciones en general. Este modelo tampoco hace un seguimiento del tamaño simbólico de las estructuras de datos. Una importante limitación de este enfoque es que sólo es viable para programas de primer orden.

En [CKQ+05, CNQR05] se presenta un sistema de tipos similar al de [HP99] para lenguajes orientados a objetos que caracteriza tamaños de las estructuras de datos y la cantidad de memoria requerida para ejecutar de forma segura métodos que operan sobre estas estructuras. El sistema de tipos esta compuesto por predicados que utilizan expresiones simbólicas, pertenecientes a la aritmética de Presburger, que capturan el tamaño de las estructuras de datos, el efecto de los métodos sobre las estructuras de datos que manipulan y la cantidad de memoria que los métodos requieren y liberan. Para cada método se captura, de forma conservadora, la cantidad de memoria requerida como una función del tamaño de la entrada de los métodos. Estas expresiones son verificadas mediante el uso de un type checker. A pesar de que este enfoque permite verificar las cotas sobre la utilización de memoria, no permite la inferencia de las mismas siendo el programador el responsable de su especificación. Otra debilidad de este enfoque son las expresiones que soporta ya que se limitan a

la artimética de Presburguer.

[CJPS05] presenta un algoritmo para el análisis de recursos aplicable a lenguajes con bytecode simil Java. Para un programa dado el algoritmo puede detectar métodos e instrucciones que se ejecuten una cantidad de veces no acotadas, luego puede determinar si la memoria esta acotada o no. Si bien esta técnica permite verificar que un programa requiere una cantidad de memoria acotada, no permite verificar cuando una cantidad dada es adecuada o no.

La técnica presentada por Unnikrishnan et al. [USL03] calcula requerimientos de memoria considerando *grabage collection*. Consiste en la transformación de un programa tal que, dado una función, construye una nueva función que imita de manera simbólica la memoria reservada por el primero. La función obtenida debe ser ejecutada sobre una valuación de los parámetros para obtener las cotas de asignación de memoria. La evaluación de la función de estimación puede no terminar, incluso si el programa lo hace. Por otro lado el costo de la evaluación puede ser alto y difícil de predecir de antemano.

En [BFGY08] dado un método $m$ con parámetros $p_1, ....p_k$ se obtiene una cota superior paramétrica de la cantidad de memoria necesaria para ejecutar de manera segura $m$ y todos los métodos a los que este invoca. Una característica importante de este enfoque es que las expresiones obtenidas son paramétricas y fáciles de evaluar.

Para calcular esa estimación se considera la liberación de memoria que puede ocurrir durante la ejecución del método adoptando un modelo de administración de memoria basado en regiones. Esta técnica requiere del conocimiento de las distintas configuraciones del stack de llamadas del método en análisis, en consecuencia el análisis no es modular. Las cotas de memoria obtenidas están restringidas a funciones polinomiales. Esta técnica no permite reutilizar especificaciones, tampoco es posible especificar el comportamiento de los métodos no analizables (por ejemplo, métodos recursivos).

Recientemente Albert et al. [AAG+07] proponen una técnica para el análisis paramétrico de programas secuenciales Java. El código es traducido a una representación recursiva con una pila aplanada. Luego, infieren *relaciones de tamaño* similares a los invariantes lineales utilizados en [BFGY08]. Usando estas relaciones, y la representación recursiva del programa, calculan *relaciones de costo* que son conjuntos de ecuaciones en función de los parámetros. Aplicado al consumo de memoria, las cotas que esta técnica puede inferir no son limitadas a polinomios. Sin embargo, la resolución de ecuaciones recurrentes no es una tarea trivial y no siempre es posible encontrar una solución. Tampoco permite la especificación de costos para aquellas partes del programa que resulten no analizables. Esta técnica no considera la liberación de memoria.

Al momento de iniciar la investigación, no encontramos trabajos que presenten una técnica modular para la inferencia de requerimientos de memoria dinámica. En

[CR07] se presenta un análisis estático que permite inferir resumenes de métodos que especifican los efectos producidos por un método en el *heap*. Este trabajo está enfocado en inferir los efectos sobre la creación de nuevos objetos o nuevas relaciones entre objetos como resultado de la ejecución de un método. Sin embargo no permite inferir cotas en el uso de memoria. Por otro lado, el esquema de inferencia es similar al propuesto en esta tesis, el cual consiste en la extracción de un resumen del comportamiento de un método de forma bottom-up utilizando los resumenes ya calculados para la síntesis del resumen del método bajo análisis.

Finalmente, vale la pena mencionar, el trabajo desarrollado por [Tab09] donde se presenta la implementación de una máquina virtual Java que utiliza un administrador de memoria por regiones. Si bien, este trabajo no está enfocado en la inferencia de requerimientos de memoria, la implementación soporta la especificación de regiones de tamaño fijo, cuyo espacio puede ser especificado por una expresión paramétrica en función de los parámetros del método asociado a la región. Esto permite un algoritmo de administración eficiente.

## 1.5. Estructura

En el capítulo 2 se presentan conceptos necesarios para el desarrollo de esta tesis. En particular, se presenta el problema de administración de memoria dinámica predecible en sistemas embebidos y la formalización de problemas relacionados a la optimización de programas.

El capítulo 3 presenta una técnica general para la síntesis de fórmulas no lineales [BGY06, BFGY08] que estiman de forma conservadora la cantidad de memoria explícitamente reservada por un método en función de sus parámetros. Esta técnica es utilizada como base para el desarrollo de un algoritmo composicional para la inferencia de requerimientos de consumo.

El capítulo 4 presenta una técnica composicional para el cálculo de objetos reservados por un programa. A su vez, se considera un esquema de administración por regiones y se describe una técnica que permite inferir la cantidad de objetos necesarios para la ejecución de un método teniendo en cuenta la recolección de objetos.

En el capítulo 5 se presentan las diferentes técnicas aplicadas para la resolución de los problemas que el análisis composicional requiere. A su vez, se presentan las limitaciones que las técnicas utilizadas imponen sobre el análisis.

En el capítulo 6 se presentan los detalles más relevantes relacionados a la implementación de la herramienta desarrollada para el estudio de consumo de memoria.

Los capítulos 7 y 8 presentan los experimentos realizados, conclusiones y trabajos a futuros.

Finalmente los apéndices A, B presentan documentación técnica de las herramientas desarrolladas.

# Capítulo 2

# Definiciones preliminares

En este capítulo se presentan conceptos utilizados a lo largo del desarrollo de esta tesis.

## 2.1. Cuantificando el uso de memoria

Algunas preguntas que aparecen con frecuencia en el campo de análisis y optimización de programas son:

1. ¿Cuántas operaciones son realizadas en un ciclo?
2. ¿Cúanta memoria es requerida por una parte del código?
3. ¿Cuántos elementos diferentes de un arreglo son accedidos antes que el elemento (x,y) sea accedido?

Muchas de estas respuestas son piedras angulares para la transformación y optimización de programas como el incremento del paralelismo, minimización del uso de memoria, inferencia de requerimientos de memoria, estimación del peor tiempo de ejecución, etc.

Estos problemas pueden reducirse a contar soluciones de un conjunto de restricciones. Si el sistema es lineal entonces el problema es reducido a la enumeración de soluciones enteras de un sistema lineal de inecuaciones convirtiendose la pregunta en: ¿cuántos puntos enteros $x \in Z^d$ satisfacen $Ax >= B$?.

Mas aún, muchos análisis requieren que la respuesta sea en función de un conjuto de parámetros $p$, luego el número de puntos enteros en el siguiente conjunto debe ser calculado: $P_p = \{x \in Q^d | Ax >= Bp + c\}$, donde $p$ es un vector de parámetros, $A$ y $B$ son matrices de enteros y $c$ es un vector de enteros.

**Definición** $P_p$ es llamado un politopo paramétrico cuando el número de enteros que satisfacen $P_p = \{x \in Q^d | Ax >= Bp + c\}$ es finito para cada valor de $p$. □

```
public class Counting {

  public void nObjects(int n) {
    for(int i=1;i<=n;i++) {
      for(int j=1;j<=n;j++) {
        Object object = new Object();
      }
    }
  }

  public void counting(int k) {
    for(int i=0;i<k;i++) {
      this.nObjects(i);
    }
  }
}
```

Listing 2.1: Preliminares. Ejemplo

### 2.1.1. Cantidad de soluciones de un conjunto de restricciones

**Definición** Sea $\mathcal{I}$ un conjunto de restricciones sobre un conjunto de variables enteras $V = W \uplus P$ donde $P$ representa un conjunto de variables dinstiguidas llamadas parámetros y $W$ el resto de las variables que aparecen en el conjunto de restricciones. Escribimos $\mathbf{v}$, $\mathbf{p}$ y $\mathbf{w}$ para indicar valuaciones a las variables. $\mathcal{I}(\mathbf{v})$ es el resultado de evaluar $\mathcal{I}$ en $\mathbf{v}$. $\mathcal{C}(\mathcal{I}, P)$ denota una expresión simbólica sobre $P$ que determina el *número de soluciones enteras* de $\mathcal{I}$ para el conjunto de variables $W$, asumiendo que $P$ tiene valores fijos. Más precisamente:

$$\mathcal{C}(\mathcal{I}, P) \quad = \quad \lambda \mathbf{p}.\ \#\{\ \mathbf{w} \in \mathbb{Z}^{|W|} \mid \mathcal{I}(\mathbf{w}, \mathbf{p})\ \}$$

□

**Ejemplo** Supongamos que queremos determinar la cantidad de veces que se ejecuta la instrucción *new Object* en el ejemplo. Luego obtenemos conjunto de restricciones que determinan el espacio de iteración sobre esa instrucción:

$$\mathcal{I} = \{1 \leq i \leq n,\ 1 \leq j \leq n\}$$

Definimos el conjunto de variables y parámetros del conjunto de restricciones:

$$W = \{i, j\}$$

$$P = \{n\}$$

| $\mathcal{I}$ | P | $\mathcal{C}(\mathcal{I}, P)$ |
|---|---|---|
| $\{k = i\}$ | $\{k\}$ | 1 |
| $\{1 \leq i \leq n,\ 1 \leq j \leq n\}$ | $\{n\}$ | $n^2$ |
| $\{1 \leq i \leq k\}$ | $\{k\}$ | $k$ |
| $\{1 \leq i \leq 5\}$ | $\{k\}$ | 5 |
| $\{1 \leq i \leq k, n = i, 1 \leq j \leq n\}$ | $\{k\}$ | $\frac{1}{2}k^2 + \frac{1}{2}k$ |

Cuadro 2.1: Enumeración de un conjunto de restricciones.

$$V = W \uplus P$$

Una vez fijado $n$, las variables $i$ y $j$ pueden variar entre 1 y $n$ manteniendo la restricción $\mathcal{I}$, luego existen $n^2$ valuaciones de $W$ que satisfacen $\mathcal{I}$.

$$\mathcal{C}(\mathcal{I}, P) = n^2$$

□

La tabla 2.1 presenta algunos invariantes asociados con la expresión paramétrica que determina la cantidad de soluciones enteras de cada uno. Existen diferentes técnicas que permiten calcular estas expresiones paramétricas[Cla96, Fah98].

### 2.1.2. Maximización paramétrica

Para algunas técnicas de análisis no sólo estamos interesados en el número de enteros que satisfacen un conjunto de restricciones, sino también en la máxima valuación de una función en los puntos que satisfacen el conjunto de restricciones. Por ejemplo, querríamos conocer cuál es el peor caso de ejecución de un método en el contexto de un ciclo o la mayor cantidad de memoria requerida por una invocación.

Más aún, algunos análisis requieren que el resultado de este problema de maximización sea expresado en función de los paramétros del método bajo análisis.

**Definición** Sea $\mathcal{I}$ un conjunto de restricciones sobre un conjunto de variables enteras $V = W \uplus P \uplus P'$ donde $P$ y $P'$ representan conjuntos de variables dinstiguidas (parámetros) y $W$ el resto de las variables que aparecen en el conjunto de restricciones. Sea $\mathcal{F}_{P'}$ una expresión paramétrica en función de los parámetros $P'$. Escribimos $\mathbf{v}$, $\mathbf{p}$, $\mathbf{p'}$ y $\mathbf{w}$ para indicar valuaciones a las varaibles. $\mathcal{I}(\mathbf{v})$ es el resultado de evaluar $\mathcal{I}$ en $\mathbf{v}$. $\mathcal{M}(\mathcal{I}, \mathcal{F}_{P'}, P)$ denota una expresión simbólica sobre P que provee la máxima valuación de $\mathcal{F}_{P'}$ sobre todas las valuaciones de $W$ que satisfacen el conjunto de restricciones $\mathcal{I}$. Más precisamente:

$$\mathcal{M}(\mathcal{I}, \mathcal{F}_{P'}, P) \quad = \quad \lambda \mathbf{p}.\ \text{máx}\{\ \mathcal{F}_{P'}(\mathbf{p'}) \quad | \quad \exists w \quad \mathcal{I}(w, p, p')\ \}$$

□

**Ejemplo** Para el código 2.1, la ejecución de la instrucción *this.nObjects(i)* produce la creación de $n^2$ objetos. Supongamos que queremos saber cuál es la mayor cantidad de objetos creados a lo largo del ciclo presente en el método *counting*. Este problema de maximización queda caracterizado por:

$$\mathcal{I} = \{1 \leq i \leq k, i = n\}$$

$$\mathcal{F}(n) = n^2$$

$$W = \{i, n\}$$

$$P = \{k\}$$

$$P' = \{n\}$$

$$V = W \uplus P \uplus P'$$

Este caso es fácil de determinar, ya que $\mathcal{F}$ es una función creciente y cada iteración realiza una llamada con el valor de $i$ que es también creciente. Luego el máximo queda determinado por el valor máximo de $i$ el cual está acotado por el parámetro $k$:

$$\mathcal{M}(\mathcal{I}, \mathcal{F}, P) = k^2$$

□

La tabla 2.2 presenta el resultado de maximizar expresiones paramétricas $\mathcal{F}$ sobre todas las valuaciones posibles de $\mathcal{F}$ determinadas por un conjunto de restricciones (*invariante*).

### 2.1.3. Contando soluciones con peso

Para algunas técnicas es importante conocer la suma de una expresión paramétrica evaluada en los puntos que satisfacen un conjunto de restricciones. Por ejemplo, para determinar la cantidad de memoria reservada por una invocación a un método a lo largo de un ciclo, debe tomarse en cuenta la memoria solicitada por este método en cada iteración del ciclo.

| $\mathcal{I}$ | $P$ | $\mathcal{F}$ | $\mathcal{M}(\mathcal{I},\mathcal{F},P)$ |
|---|---|---|---|
| $\{1 \leq i \leq n\}$ | $\{n\}$ | 5 | 5 |
| $\{1 \leq i \leq n, i = k\}$ | $\{n\}$ | 3k | $3n$ |
| $\{1 \leq i \leq n, i = k\}$ | $\{n\}$ | $k^2 + k$ | $3n^2 + n$ |
| $\{1 \leq i \leq k, i = n\}$ | $\{k\}$ | $n$ | $k$ |
| $\{k = p, 1 \leq i \leq k, i = n, 1 \leq j \leq k\}$ | $\{p\}$ | $n$ | $\frac{1}{2}p^2 + \frac{1}{2}p$ |

Cuadro 2.2: Maximización de una expresión paramétrica sobre un conjunto de restricciones.

**Definición** Sea $\mathcal{I}$ un conjunto de restricciones sobre un conjunto de variables enteras $V = W \uplus P \uplus P'$ donde $P$ y $P'$ representan conjuntos de variables dinstiguidas (parámetros) y $W$ el resto de las variables que aparecen en el conjunto de restricciones. Sea $\mathcal{F}_{P'}$ una expresión paramétrica en función de los parámetros $P'$. Escribimos $\mathbf{v}$, $\mathbf{p}$, $\mathbf{p}$ y $\mathbf{w}$ para indicar valuaciones a las varaibles. $\mathcal{I}(\mathbf{v})$ es el resultado de evaluar $\mathcal{I}$ en $\mathbf{v}$. $\mathcal{CP}(\mathcal{I},\mathcal{F}_{P'},P)$ denota una expresión simbólica sobre P que provee la suma de evaluar $\mathcal{F}P'$ sobre todas las valuaciones de $W$ que satisfacen el conjunto de restricciones $\mathcal{I}$. Más precisamente:

$$\mathcal{CP}(\mathcal{I},\mathcal{F},P) \quad = \quad \lambda\mathbf{p}. \sum_{p' \mid \exists w\ \mathcal{I}(w,p,p')} \mathcal{F}_{P'}(p')$$

□

**Ejemplo** Supongamos que queremos determinar la cantidad total de objetos creados por el método *counting* en el código 2.1. Para esto debemos sumar la cantidad de objetos creados por la invocación al método *nObjects* en cada iteración. Este problema queda caracterizado por:

$$\mathcal{I} = \{1 \leq i \leq k, i = n\}$$

$$\mathcal{F} = n^2$$

$$W = \{i, n\}$$

$$P = \{k\}$$

$$V = W \uplus P$$

Podemos observar que cada iteración implica la creación de $i^2$ objetos. Luego, la cantidad total de objetos creados es:

$$\begin{aligned} \mathcal{CP}(\mathcal{I}, \mathcal{F}, P) &= \sum_{i=1}^{k} i^2 \\ &= \frac{1}{3}k^3 + \frac{1}{2}k^2 + \frac{1}{6}k \end{aligned}$$

□

La tabla 2.3 presenta el resultado de sumar una expresión paramétrica sobre un dominio.

Para funciones polinomiales, restringiendo el dominio a un conjunto de restricciones lineales, existen distintos algoritmos que permiten calcular estas expresiones paramétricas [VB08].

| $\mathcal{I}$ | $P$ | $\mathcal{F}$ | $\mathcal{CP}(\mathcal{I}, \mathcal{F}, P)$ |
|---|---|---|---|
| $\{1 \leq i \leq n\}$ | $\{n\}$ | 5 | $5n$ |
| $\{1 \leq i \leq n,\ 1 \leq j \leq n\}$ | $\{n\}$ | 5 | $5n^2$ |
| $\{1 \leq i \leq n,\ 1 \leq j \leq n, i = k\}$ | $\{n\}$ | $k$ | $\frac{1}{2}n^2 + \frac{1}{2}n$ |
| $\{1 \leq i \leq p, n = i, m = q\}$ | $\{p, q\}$ | $n + m$ | $\frac{1}{2}p^2 + \frac{1}{2}p + p \cdot q$ |
| $\{1 \leq i \leq k, i = p\}$ | $\{p\}$ | $p^2$ | $\frac{1}{3}p^3 + \frac{1}{2}p^2 + \frac{1}{6}p$ |

Cuadro 2.3: Suma de una expresión paramétrica sobre un dominio

## 2.2. Notación para programas

Un programa es un conjunto $\{m_0, m_1, \ldots\}$ de métodos. Un método posee una lista $P_m$ de parámetros ($\mathbf{p_m}$ denotará los argumentos del método cuando $m$ es llamado por otro método $m'$) y una secuencia de sentencias. Se asume que los parámetros del método son de tipo entero, que no hay conflicto de nombres incluyendo parámetros formales, nombres de variables locales y globales y que no hay recursión.

Cada sentencia del programa es asociada con una *ubicación de control* $\ell = (m, n) \in \mathit{Label} =_{def} \mathit{Method} \times \mathbb{N}$ (un método y una posición dentro del método) que caracterizan de manera única la sentencia a través de un mapping stm ($\mathit{stm} : \mathit{Label} \to \mathit{Statement}$).

## 2.3. Organizaciones de memoria predecibles

Los lenguajes orientados a objetos, como Java, proveen una administración de memoria automática (garbage collector). Sin embargo este tipo de administración de

memoria no es utilizado para sistemas embebido de tiempo real. La principal razón es que el comportamiento temporal del software que libera la memoria dinámica es extremadamente difícil de predecir.

Diferentes algoritmos de garbage collector han sido propuestos para el uso en sistemas embebidos de tiempo real. Por ejemplo, [Hen98] propone usar un algoritmo de copia incremetal [Bro84] durante la ejecución de tareas de baja prioridad. Debe preasignarase suficiente memoria para asegurar que las tareas de alta prioridad no se queden sin espacio. Además la relación del uso compartido del tiempo con las tareas de menor prioridad no es evidente.

[Sie00] adapta el algoritmo incremental *mark and sweep* para una JVM que reserva objetos como una colección de pequeños bloques. El inconveniente de este algoritmo es que el número de incrementos requeridos por bloque asignado depende en el tamaño de toda la memoria alcanzable.

[RF02] adapta el clásico *reference counting*. Su tiempo de respuesta depende del número total de objetos alcanzables cuando debe liberar un ciclo no referenciado.

Una solución atractiva al problema de recolección de memoria dinámica es asignar objetos en regiones [TT97]. En un esquema de memoria basado en regiones los objetos con tiempos de vida similares son asociados a la misma área de memoria, la cual puede ser liberada en su totalidad cuando los objetos incluídos ya no son requeridos. Luego la asignación y liberación de objetos puede ser realizada en un tiempo predecible.

### 2.3.1. Organización por regiones

Este esquema, utilizado por *Real-Time Specification for Java (RTSJ)* [BG00], permite al programador especificar que una unidad de cómputo debe correr en el contexto de una región preasignada. Sin embargo programar utilizando la *RTSJ* es usualmente más complejo que utilizar la versión estandard de Java [PFHV04]. El programador debe decidir en que región deben asignarse sus estructuras de datos y asignar tamaños a las regiones lo que impone cierta dificultad.

En vez de requerir al programador que decida donde asignar los objetos, es posible utilizar técnicas de análisis estático para determinar de forma automática la ubicación de los objetos. De esta manera el programa puede ser transformado de forma transparente reemplazando las instrucciones de tipo *new* (creación de objetos) por una invocación al administrador de la región elegida.

Este enfoque requiere el cálculo del tiempo de vida de los objetos dinámicamente creados con el objeto de insertar llamadas al componente responsable de liberar la región tan pronto como todos los objetos incluídos no sean requeridos.

### 2.3.2. Análisis de escape

El objetivo del análisis es determinar el alcance de un objeto. Esta técnica es usada en diferentes aplicaciones como remoción de sincronizaciones, eliminación de chequeos en tiempo de ejecución, asignación de objetos en la pila (*stack allocation*), etc.

Por ejemplo, [GS00] propone el uso de esta técnica para la asignación de objetos en la pila. El resultado del análisis permite, entre otras aplicaciones, utilizar el espacio de memoria de la pila para la creación de objetos que no escapan a un método.

```
public class Escape {

    public Integer[] shuffle(int items) {
        1: Date now = new Date();
        2: Random random = new Random(now.getTime());
        3: Integer[] shuffle = new  Integer[items];
        4: for(int i = 0; i < items; i ++) {
        5:       shuffle[i] = new Integer(random.nextInt());
            }
        6: return shuffle;
    }
}
```

Listing 2.2: Analisis de escape.

Consideremos a modo de ejemplo el código 2.2 donde el método $shuffle$ retorna una lista de enteros aleatorios. Es posible determinar en tiempo de compilación que el objeto *now* creado en la ubicación de control $shuffle.1$ y el objeto *random* ($shuffle.2$) no escapan al método *suffle* y en consecuencia puede ser almacenado en la pila. Por otro lado el objeto *shuffle* creado en la ubicación de control $shuffle.5$ es retornado, luego el tiempo de vida de este objeto es mayor al del método y consecuentemente no puede ser alocado en el stack del método.

En este trabajo, nos interesa determinar de forma conservadora cuándo un objeto escapa o es capturado por un método.

**Definición** Un objeto escapa cuando su tiempo de vida es mayor al tiempo de vida del método. □

# Capítulo 3

# Cálculo de consumo de memoria

En este capítulo presentamos la técnicas desarrolladas en [BGY06, BFGY08] para el cálculo de cotas paramétricas sobre la cantidad de memoria dinámicamente reservada por programas Java.

El análisis presentado en [BGY06] consiste en cuantificar los pedidos de memoria dinámica realizados por un método. Dado un método $m$ de parámetros $P_m$ se presenta un algoritmo que calcula un polinomio en términos de $P_m$ que aproxima (cota superior) la cantidad de memoria reservada durante la ejecución de $m$. Esta cota es una aproximación simbólica de la cantidad total de memoria pedida por la aplicación a la máquina virtual mediante instrucciones *new*. Básicamente, la técnica identifica los puntos de creación (instrucciones *new*) alcanzables desde el método bajo análisis y cuenta el número de veces que estas instrucciones son visitadas. Para esto, se consideran invariantes que describen las posibles valuaciones de las variables en cada punto de creación y se cuenta el número de soluciones enteras de estos invariantes. Finalmente, el resultado es adaptado para considerar el tipo de cada objeto reservado.

Posteriormente en [BFGY08] se presenta una técnica para aproximar la cantidad de memoria requerida para ejecutar un método considerando la liberación de memoria. Dado un método $m$ de parámetros $P_m$ se obtiene una cota superior paramétrica de la cantidad de memoria necesaria para ejecutar de manera segura un método y todos los métodos que este llama sin agotar la memoria. Esta expresión puede ser vista como una precondición que establace que $m$ requiere esa cantidad de memoria libre antes de su ejecución. Para calcular esa estimación se considera la liberación de memoria que puede ocurrir durante la ejecución de un método en un modelo de administración de memoria por regiones (ver 2.3.1). Luego, se modelan las potenciales configuraciones de la pila de regiones para determinar el máximo consumo del método bajo análisis. Este modelo lleva a un conjunto de problemas de maximización de polinomios, resuelto mediante una técnica basada en bases de Bernstein [CFGV09] que determina una solución paramétrica.

Una importante característica de este enfoque es que las expresiones obtenidas son paramétricas y fáciles de evaluar.

En las siguientes secciones se presentan los detalles de ambas técnicas. El código 3.1 será utilizado como ejemplo a lo largo del capítulo.

```
    public void m0(int mc) {
        1: m1(mc);
        2: B[] m2Arr = m2(2*mc);
        3: B[] m3Arr = m3(mc);
    }

    private void m1(int k) {
           for(int i = 1; i <= k; i++) {
        4:     A a = new A();
        5:     B[] dummyArr = m2(i);
           }
    }

    private B[] m2(int n) {
        6:  B[] arrB = new B[n];
        7:   for(int j = 1; j < n; j++) {
        8:     arrB[j-1] = new B();
        9:     C c = new C();
        10: c.value = arrB[j-1];
           }
        11: return arrB;
    }

    private B[] m3 (int k) {
        12: B[] bArr;
           for(int j = 1; j <= k; j++) {
        13:     bArr = m2(i);
        14: C c = new C();
           }
        15: return  bArr;
    }
}
```

Listing 3.1: Consumo. Ejemplo

## 3.1. Memoria reservada por un método

Para calcular la cantidad de memoria reservada por un método, se indentifican los puntos de creación alcanzables desde el método bajo análisis y se suma la cantidad de memoria utilizada por cada punto.

Un punto de creación es identificado como una cadena de llamadas que comienza en el método bajo análisis y termina en una instrucción del tipo *new*.

**Definición** Un punto de creación $cs$ es una secuencia $\pi.\ell$, donde $\ell$ es una ubicación de control y $\pi$ es un camino al método $\mathsf{mth}(\ell)$ en el grado de llamadas $G$.

**Ejemplo** $cs = m0.1.m1.5.m2.9$ es un punto de creación. □

La cantidad de memoria reservada por un punto de creación $cs$ puede ser calculada caracterizando el espacio de iteración de dicho punto. Esto puede ser relacionado con el número de soluciones enteras de un predicado que restringe las valuaciones de las variables para la ubicación de control de $cs$ (un invariante). Siendo que un punto de creación es representado como un camino a través de varios métodos, se utilizan invariantes globales. Este problema fue presentado en la sección 2.1.1).

Sea $CS_m \subseteq CS$ el conjuntos de puntos de creación alcanzables desde $m$.

**Ejemplo** Los puntos de creación para el ejemplo 3.1 son:

$$
\begin{aligned}
CS_{m0} = \; & \{\, m0.1.m1.4,\ m0.1.m1.5.m2.6,\ m0.1.m1.5.m2.8, \\
& m0.1.m1.5.m2.9,\ m0.2.m2.6,\ m0.2.m2.8,\ m0.2.m2.9 \\
& m0.3.m3.13.m2.6,\ m0.3.m3.13.m2.8,\ m0.3.m3.13.m2.9, \\
& m0.3.m3.14 \,\} \\
CS_{m1} = \; & \{\, m1.4,\ m1.5.m2.6,\ \ m1.5.m2.8,\ \ m1.5.m2.9 \,\} \\
CS_{m2} = \; & \{\, m2.6,\ m2.8,\ m2.9 \,\} \\
CS_{m3} = \; & \{\, m3.14,\ m3.13.m2.6,\ m3.13.m2.8,\ m3.13.m2.9 \,\}
\end{aligned}
$$

□

El problema de estimar la cantidad de memoria reservada por un método $m$ puede ser reducido a:

1. Identificar los puntos de creación alcanzables de $m$
2. Por cada $cs \in CS_m$, obenter el invariante $\mathcal{I}^m_{cs}$.
3. Contar el número de soluciones enteras del invariante en términos de los parámetros de $m$ determinado por $\mathcal{C}(\mathcal{I}^m_{cs}, P_m)$. Estas expresiones son adaptadas para considerar el tamaño de los objetos reservados.
4. Sumar los resultados.

La función *computeAlloc* determina una expresión paramétrica en función de los parámetros del método que aproxima de forma conservadora la cantidad de memoria para un conjunto de puntos de creación:

$$computeAlloc(m, CS) = \sum_{cs \in CS} \mathcal{C}(\mathcal{I}^m_{cs}, P_m) \text{ donde } CS \subseteq CS_m$$

Dado un método $m$, el estimador simbólico de la memoria dinámicamente reservada por $m$ está definido por:

$$memAlloc(m) = computeAlloc(m, CS_m)$$

## 3.2. Memoria reservada en un esquema por regiones

En [BFGY08] se propone la utilización de un esquema de administración basado en regiones (ver sección 2.3.1) asociando a cada método una región en donde se alojan los objetos capturados por el método. Al finalizar la ejecución de un método la región asociada es recolectada junto a todos los objetos alojados en ésta.

Para caracterizar la cantidad de memoria que escapa y la cantidad de memoria capturada para un método $m$ se utiliza la función *computeAlloc* (descripta en la sección 3.1) restringiendo el conjunto de puntos de creación a aquellos que escapan o son capturados. Estos filtros son calculados por las funciones $escape(m)$ y $capture(m)$ que son definidas utilizando un análisis de escape.

$$memEscapes(m) = computeAlloc(m, escape(m))$$

Este cuantificador permite conocer la cantidad de memoria reservada por un método en regiones activas luego de que la región asignada al método es liberada. Es, a su vez, un indicador de la cantidad de memoria que no puede ser recolectada por el administrador de memoria luego que finaliza la ejecución del método.

$$memCaptured(m) = computeAlloc(m, capture(m))$$

La expresión resultante es un estimador simbólico del tamaño de la región de memoria asociada al método.

### 3.2.1. Memoria necesaria para la ejecución de un método

Para calcular la memoria requerida por un método $m$ no sólo es necesario conocer el tamaño de la región asociada sino que debe considerarse el tamaño de las regiones de los métodos que pueden ser invocados durante la ejecución de $m$. Siendo que la instanciación de regiones está asociada a la invocación de los métodos la pila de regiones estará relacionada con caminos en el árbol de llamadas de la aplicación. Hay dos hechos importantes que deben ser tenidos en cuenta:

- Hay algunas configuraciones de pilas de regiones que nunca podrán ocurrir al mismo tiempo.

- Dado un camino $\pi$ del árbol de llamadas siempre habrá un pila de regiones de longitud $|\pi|$ pero el tamaño de cada región puede variar en función de los parámetros asignados cada vez que un método es invocado.

Consideremos, con el objeto de ilustrar el primer hecho, el método $m0$ en el ejemplo 3.1. En la ubicación $m0.1$ se invoca a m1 que a su vez invoca a m2. Por otro lado en la ubicación $m0.2$, m0 invoca a m2. En el esquema de administración de memoria por regiones descripto, existirán tres regiones activas para la cadena de llamadas $m0 \rightarrow m1 \rightarrow m2$ y dos regiones activas para la cadena de llamadas $m0 \rightarrow m2$ (ver figura 3.1). Estas dos cadenas son independientes dado que no pueden estar simultáneamente activas.

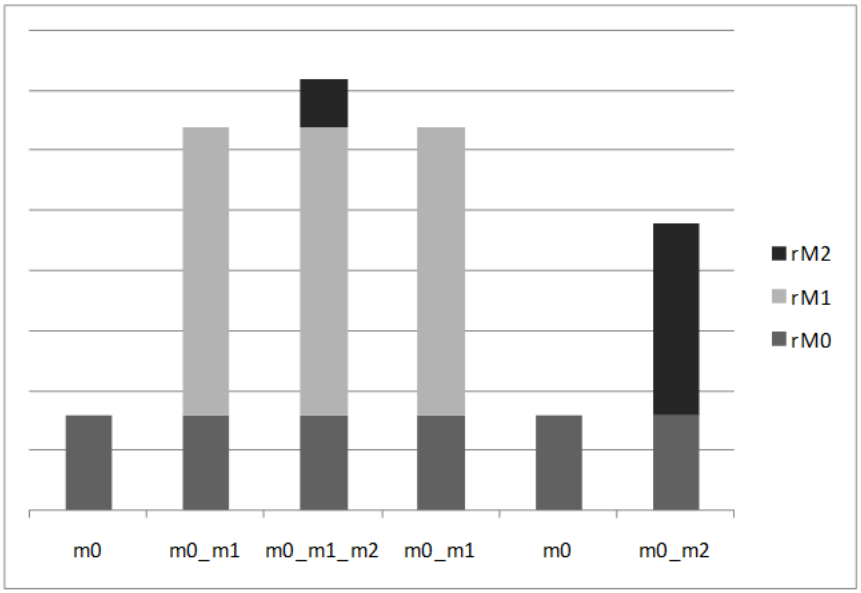


Figura 3.1: Regiones para la cadena de llamadas

Para ilustrar el segundo hecho, consideremos la cadena de llamadas $m0.1 \rightarrow m1.5 \rightarrow m2$. El método $m2$ será invocado $k$ veces con el parámetro $n$ variando entre 1 y $k$. Por cada invocación a $m2$ una nueva región es creada, la cual es recolectada al finalizar la ejecución de $m2$. Dado que habrá una sola región activa para $m2$ es suficiente con considerar el mayor tamaño que la región puede adoptar de acuerdo al contexto de llamada. En este caso el tamaño de la región alzanca el mayor tamaño cuando $n = k$.

Para determinar la cantidad de memoria necesaria para ejecutar de manera segura un método es suficiente con considerar para cada potecial camino en el árbol de llamadas comenzando desde el método bajo análisis ($mua$) el mayor tamaño que las regiones pueden adoptar en el contexto de llamada.

Sea $MaxRegSize_{mua}^{\pi.m}$ una función que determina una expresión, en función de los parámetros de $mua$, del tamaño de la mayor región creada por cualquier llamada

a $m$ con la pila de control $\pi$ en un programa que comienza en $mua$. Suponiendo que es posible calcular $MaxRegSize$ para cada método en cada cadena de llamadas entonces para calcular la cantidad de memoria requerida para ejecutar un método $mua$, hace falta considerar el tamaño de su propia región y sumar la cantidad de memoria requerida por la ejecución de cada uno de los métodos que invoca. Como cada llamada implica una pila de regiones independiente es posible seleccionar la rama que requiere la mayor cantidad de memoria. Este procedimiento se aplica de manera recursiva a través del árbol de llamadas. Esta función puede ser definida como:

$$\begin{array}{l} memRq_{mua}^{\pi \cdot m}(p_{mua}) = MaxRegSize_{mua}^{\pi.m}(p_{mua}) + \\ \text{máx}\{\ memReq_{mua}^{\pi.m.l.m_i}(p_{mua}) \mid (m, l, m_i) \in edges(CG_{mua} \downarrow \pi.\, m)\ \} \end{array}$$

donde $CG_{mua} \downarrow \pi.\, m$ es la proyección sobre el camino $\pi.\, m$ del árbol de llamadas del programa comenzando en el método $mua$ y $edges$ es el conjunto de sus aristas.

Notar que para definir correctamente $memRq$ es necesario descartar las llamadas recursivas.

Finalmente, para predecir de manera segura la cantidad de memoria requerida por $mua$, hace falta considerar los objetos reservados durante su ejecución que no pueden ser recolectados cuando finaliza. Luego, la función que aproxima los requerimientos de memoria se define como:

$$memRq_{mua}(p_{mua}) = memEscapes(mua)(p_{mua}) \ + \ memReq_{mua}^{mua}(p_{mua})$$

**Ejemplo** La memoria requerida para ejecutar $m0$ en 3.1 es:

$$\begin{array}{rcl} memRq_{m0}(mc) & = & memEscapes(m0)(mc) + \text{MaxRegSize}_{m0}^{m0}(mc) \\ & + max\{ & \text{MaxRegSize}_{m0}^{m0.1.m1}(mc) + \text{MaxRegSize}_{m0}^{m0.1.m1.5.m2}(mc), \\ & & \text{MaxRegSize}_{m0}^{m0.2.m2}(mc), \\ & & \text{MaxRegSize}_{m0}^{m0.3.m3}(mc) + \text{MaxRegSize}_{m0}^{m0.3.m3.13.m2}(mc)\ \} \end{array}$$

□

### Maximizando el tamaño de regiones

Como se menciona en la sección anterior, es necesario modelar el hecho de que el tamaño de una región puede variar según el contexto de llamada. Luego, para cada método $m'$ alcanzable desde el método bajo análisis hace falta determinar una expresión que represente el mayor tamaño que una región para $m'$ puede alcanzar restringido a una cadena de llamadas $\pi$ empezando en $mua$. Para relacionar los parámetros de $m'$ con los parámetros de $mua$ y para restringir las valuaciones de las

variables de acuerdo al contexto de llamadas se utilizan invariantes. Sea $mua \dots m'$ un camino comenzando en $mua$ que termina en $m'$ se modela el tamaño máximo de región como:

$$\begin{aligned} \text{MaxRegSize}_{mua}^{mua \dots m'}(P_{mua}) &= \text{Maximize } memCaptured(m')(P_m) \\ &\quad \text{sujeto a } I_{mua}^{mua \dots m'}(P_{mua}, P_m, W) \end{aligned}$$

$memCaptured(m')$ es un polinomio en términos de los parámetros de $m'$ y el invariante $\mathcal{I}_{mua}^{mua \dots m'}$ liga los parámetros de $m'$ con los parámetros de $mua$.

Esta fórmula caracteriza un problema de maximización no lineal cuya solución es una expresión en función de los parámetros de $mua$. Para resolver este problema de maximización se utiliza un enfoque basado en una extensión de la expansión de Bernstein [CFGV09] para manipular polinomios paramétricos en muchas variables.

## 3.3. Conclusiones

En esta capítulo presentamos una técnica para aproximar la cantidad total de memoria dinámicamente reservada por un programa [BGY06]. Dado un método $m$ con parámetros $P_m$ se obtiene una cota superior paramétrica en términos de $P_m$ de la cantidad de memoria reservada por la ejecución de $m$ y todos los métodos a los que éste invoca.

A su vez, considerando la liberación de memoria que puede ocurrir durante la ejecución de $m$, se determina una cota superior de la cantidad de memoria necesaria para la ejecución segura de $m$ [BFGY08].

Esta técnica calcula efectivamente requerimientos de memoria. Para esto requiere del conocimiento de las distintas configuraciones de la pila de llamadas del método en análisis. Es decir, el método requiere conocer su contexto de llamada, condición que en general no es deseable. Por otro lado utiliza el concepto de punto de creación como un camino en el árbol de llamadas lo que limita el análisis a programas no recursivos.

Tomando como base las técnicas presentadas en [BGY06, BFGY08] pero modificando el enfoque del análisis creemos que la técnica puede ser fortalecida. Para esto, proponemos un análisis composicional que infiere resúmenes por métodos, los cuales especifican los efectos de invocar a un método desde el punto de vista del análisis de memoria. Este nuevo enfoque permitiría la reutilización de especificaciones, mejorar la escalabilidad, mejorar la integración con otras técnicas, incrementar la capacidad de abstracción, etc.

El capítulo siguiente presenta este nuevo enfoque para el cálculo de requerimientos de memoria inspirado en la técnicas presentadas a lo largo de este capítulo.

# Capítulo 4

# Análisis composicional del consumo de memoria

En el capítulo anterior presentamos una técnica original de análisis estático para aproximar los requerimientos de memoria de un programa. Dado un método $m$ de parámetros $P_m$ la técnica permite determinar una cota superior de la cantidad total de memoria dinámicamente reservada por $m$ en términos de $P_m$. A su vez, considerando la memoria que puede ser liberada durante la ejecución de $m$ se calcula una cota superior de la cantidad de memoria necesaria para la ejecución de $m$ y de los métodos que $m$ invoca.

Esta técnica modela un punto de creación como un camino en el árbol de llamadas y por cada punto define varios caminos al considerar información de contexto. Esta información es, en general, redundante y difícil de mantener. A su vez este enfoque impone una limitación para el análisis de programas recursivos y limita la capacidad de abstracción. Vamos a mantener la condición sobre la no recursividad de los programas.

Un análisis composicional, basado en la ténica desarrollada en [BFGY08], que infiera resúmenes por método que describan el efecto de invocar al método permitiría atacar de forma natural los puntos mencionados. Además creemos que permitiría fortalecer la técnica de análisis obteniéndose mejoras en los siguientes aspectos:

- Reutilización de especificaciones, ya sea calculadas por la misma herramienta o provista por el programador.
- Mayor escalabilidad.
- Capacidad de analizar programas con métodos no analizables. Estos casos podrían ser especificados por el programador o bien podría disponerse de una especificación previa.
- Mayor capacidad de integración a otras ténicas.

La idea detrás del análisis composicional es analizar un método en dos etapas. La primera de estas tiene en cuenta el comportamiento local del método, es decir, considera todos los objetos que el método crea sin observar las llamadas que este realiza. La segunda etapa infiere los requerimientos de memoria del método analizando las llamadas que el método realiza y el resultado del análisis local.

Para analizar localmente un método, es necesario determinar el conjunto de objetos creados en el cuerpo del método bajo análisis y cuantificar este conjunto. Al igual que en [BFGY08], se utilizan técnicas de análisis estático para determinar los puntos de creación de un método, pero sin tener en cuenta los métodos llamados (no se analizan los puntos de creación alcanzables según la cadena de llamadas) y se utilizan invariantes locales para describir los espacios de iteración de estos puntos. Luego, la cantidad de visitas de un punto de creación es aproximada por la cantidad de soluciones enteras del invariante asociado.

Para analizar el efecto que producen las distintas llamadas que el método bajo análisis realiza, se define un modelo que especifica el comportamiento de un método (resumen) desde el punto de vista del consumo de memoria. Identificando cada uno de los puntos de invocación a métodos y considerando el resumen calculado para cada método invocado, se infiere el efecto sobre el *heap* que la ejecución de la llamada implica. Para esto se consideran invariantes locales que describen el espacio de iteración y ligan las variables locales del método bajo análisis con los parámetros del método llamado.

Notar que al analizar un método, se asume que ya fue calculado el resumen que modela el comportamiento de cada método llamado. Existen diferentes opciones para atacar este problema, nuestro enfoque es realizar el análisis de manera bottom-up (orden del grafo de llamadas que considera las dependencias).

El análisis propuesto considera una partición de los objetos creados durante la ejecución de un método. De esta forma podemos ver el consumo de un método $m$ de parámetros $P_m$ como $M_m(P_m) = Temp_m(P_m) + Res_m(P_m)$, donde $Temp_m(P_m)$ son los objetos que no exceden el tiempo de vida de m (temporal de m) y $Res_m(P_m)$ son los objetos que sí exceden el tiempo de vida de m (residual de m). Estas expresiones son determinadas por los objetos creados localmente por $m$ y por los objetos creados durante las llamadas realizadas por $m$.

En este nuevo enfoque debemos entonces calcular expresiones de consumo locales e inferir expresiones en función de la especificación de los métodos llamados.

Para determinar el comportamiento local de un método podemos utilizar las técnicas presentadas en [BFGY08] considerando ahora sólo invariantes locales. Esta es una diferencia importante ya que en [BFGY08] los puntos de creación son representados como un camino en el árbol de llamadas, consecuentemente los invariantes son proposiciones sobre las variables que aparecen a lo largo de ese camino, siendo difícil su comprensión y manipulación.

Para analizar las llamadas que un método realiza, se requiere considerar los invariantes en el momento de la llamada y la especificación del método invocado. Hay dos tipos de consumo que debemos tener en cuenta al momento de analizar la invocación a un método. El primero de estos son los objetos temporales del método llamado, temporales en el sentido que su tiempo de vida se limita al método llamado, para lo que es necesario disponer de memoria suficiente. El segundo tipo de consumo es el residual del método llamado, es decir, los objetos que exceden el tiempo de vida del método llamado y que viven en el contexto del método llamador.

Al examinar el efecto producido por las llamadas que un método $m$ realiza, siendo que al finalizar la ejecución del método llamado el temporal es liberado, podemos considerar el máximo temporal requerido entre todas estas. Esto asegura que para cualquier llamada que $m$ realice, la cantidad de memoria es suficiente para una ejecución segura. Por otro lado, la memoria requerida por los objetos residuales, al permanecer vivos durante la ejecución de $m$, debe ser acumulada. Esto determina una serie de problemas de maximización y sumas que presentaremos en detalle más adelante. El problema de maximización puede ser atacado siguiendo el mismo enfoque utilizado en [BFGY08]. Las sumas pueden ser calculadas mediante una técnica similar al conteo de soluciones enteras de un invariente, pero considerando una función de peso.

Por otro lado, determinar el temporal y el residual requiere calcular el tiempo de vida de los objetos. Esto es resuelto mediante el uso de una técnica de análisis de escape que será presentada en el siguiente capítulo.

Las siguientes secciones presentan el análisis composicional desarrollado a lo largo de esta tesis. Para ilustrar las distintas técnicas y definiciones presentadas recurrimos al ejemplo 3.1 utilizado en el capítulo anterior.

## 4.1. Objetos reservados por un método

En la sección 3.1 presentamos el estimador $memAlloc(m)$ que determina la cantidad total de memoria reservada por la ejecución de un método. A efectos de simplificar, y sin perdida de generalidad, vamos a considerar la cantidad de objetos creados por la ejecución de un método y no la cantidad de memoria, llamaremos a este estimador $objectsAlloc(m)$ En esta sección presentamos un cálculo composicional para obtener dicho estimador.

### 4.1.1. Objetos reservados por un punto de creación

La cantidad de objetos reservados por un punto de creación es determinada por el número de soluciones enteras de un conjunto de restricciones. Estas restricciones corresponden al invariante que determina el espacio de iteración del punto analizado. Un punto de creación queda determinado por una ubicación, y no es modelado como

un camino en el árbol de llamadas, ya que solo se consideran las instrucciones que crean objetos en el cuerpo del método bajo análisis. Consecuentemente los invariantes que se consideran son ahora locales.

**Definición** Sea un método $m$ definimos un punto de creación $lcs = \ell$ como una ubicación asociada a una instrucción de creación de objetos ($new$, $newA$) y $LCS_m$ como el conjunto de puntos de creación de $m$.

**Ejemplo** Puntos de creación para el ejemplo 3.1:

$$\begin{aligned} LCS_{m0} &= \{\ \} \\ LCS_{m1} &= \{\ m1.4\ \} \\ LCS_{m2} &= \{\ m2.6,\ m2.8,\ \ m2.9\ \} \\ LCS_{m3} &= \{\ m3.14\ \} \end{aligned}$$

□

Sea $m$ un método de parámetros $P_m$, $lcs$ un punto de creación de $m$, $\mathcal{I}^m_{lcs}$ un invariante local para $lcs$ en $m$, $\mathcal{O}(\mathcal{I}^m_{lcs}, P_m)$ calcula el número paramétrico de visitas a $lcs$. Esta expresión paramétrica estima de forma conservadora la cantidad de objetos reservados por $lcs$.

$$\mathcal{O}(\mathcal{I}^m_{lcs}, P_m) = \mathcal{C}(\mathcal{I}^m_{lcs}, P_m)$$

**Ejemplo** Consideremos el sitio de creación $m2.8$ para $m2$ en el ejemplo 3.1. El invariante $\mathcal{I}^{m2}_{m2.8}$ determina el espacio de iteración para el sitio de creación.

$$\mathcal{I}^{m2}_{m2.8} = \{\ 1 \leq j \leq n\ \}$$

$$\begin{aligned} \mathcal{O}(\mathcal{I}^{m2}_{m2.8}, P_{m2}) &= \mathcal{C}(\mathcal{I}^{m2}_{m2.8}, P_{m2}) \\ &= n \end{aligned}$$

□

### 4.1.2. Objetos reservados por la invocación a un método

Para calcular la cantidad de objetos reservados por un punto del programa en donde un método $m$ invoca a un método $m'$, debemos sumar la cantidad de objetos reservados producto de la ejecución del método $m'$. Si la invocación a $m'$ es realizada en el contexto de un ciclo, entonces debemos acumular la cantidad de objetos reservados de la llamada a lo largo del ciclo. Este problema fue caracterizado en la sección 2.1.3 como la suma de una expresión paramétrica sobre un dominio determinado por el invariante de la llamada al método $m'$.

**Definición** Sea un método $m$ definimos un punto de invocación $lis = \ell.\, m'$ como una ubicación asociada a una instrucción de invocación al método $m'$ en el contexto de $m$. $LIS_m$ representa el conjunto de puntos de invocación de $m$. Para denotar los parámetros del método llamado por $lis$ definimos la expresión $P_{lis.target} = P_{m'}$.

**Ejemplo** Puntos de invocación para el ejemplo 3.1:

$LIS_{m0} = \{\ m0.\,1.\,m1,\ m0.\,2.\,m2\ \}$

$LIS_{m1} = \{\ m1.\,5.\,m2\ \}$

$LIS_{m2} = \{\ \}$

$LIS_{m3} = \{\ m3.\,13.m2\ \}$

□

Sea $lis = \ell.\, m'$ un sitio de invocación en el contexto del método $m$ que invoca al método $m'$. Sea $\mathcal{O}_{P_{m'}}$ una expresión paramétrica, en función de los parámetros de $m'$, que determina la cantidad de objetos reservados por el método $m'$. Sea $\mathcal{I}^m_{lis}$ un invariante local que determina el espacio de iteración de $lis$ en $m$ y vincula las variables de $m$ con los parámetros $P_{m'}$. Entonces $\mathcal{O}(\mathcal{I}^m_{lis}, \mathcal{O}_{P_{m'}}, P_m)$ denota una expresión simbólica sobre $P_m$ que provee la suma de la cantidad de objetos reservados por la invocación de $m'$ en el espacio de iteración determinado por $\mathcal{I}^m_{lis}$. Más precisamente:

$$\mathcal{O}(\mathcal{I}^m_{lis}, \mathcal{O}_{P_{m'}}, P_m) = \mathcal{CP}(\mathcal{I}^m_{lis}, \mathcal{O}_{P_{m'}}, P_m)$$

**Ejemplo** El método $m2$, en el código 3.1, reserva $2n + 1$ objetos. Para el punto de invocación $lis = m1.\,5.\,m2$, en cada iteración del ciclo $m2$ es llamado con $n = i$, luego $lis$ reservará $2i + 1$ objetos en cada iteración. Podemos determinar la cantidad de objetos requeridas por $lis = m1.\,5.\,m2$ como:

$$\mathcal{I}^{m1}_{m1.5.m2} = \{1 \leq i \leq k, i = n\}$$

$$P_{m1} = \{k\}$$

$$P_{m2} = \{n\}$$

$$\mathcal{O}_{P_{m2}} = 2n + 1$$

$$
\begin{aligned}
\mathcal{O}(\mathcal{I}_{m1.5.m2}, \mathcal{O}_{P_{m2}}, P_{m1}) &= \mathcal{CP}(\mathcal{I}^{m1}_{m1.5.m2}, \mathcal{O}_{P_{m2}}, P_{m1}) \\
&= \sum_{i=1}^{k} 2i + 1 \\
&= 2 \cdot \sum_{i=1}^{k} i + \sum_{i=1}^{k} 1 \\
&= 2 \cdot \frac{(k+1) \cdot k}{2} + k \\
&= k^2 + 2k
\end{aligned}
$$

□

### 4.1.3. Total de objetos reservados por un método

Para calcular el estimador $objectsAlloc(m)$ de manera composicional se requiere determinar los puntos de creación locales de un método en vez de considerar todos los puntos alcanzables desde el mismo. Esto es calculado obteniendo todos los puntos de creación $lcs$ de $m$ y en función del invariante local asociado a cada punto se suma la cantidad de objetos reservados mediante el estimador paramétrico $\mathcal{O}(\mathcal{I}^m_{lcs}, P_m)$.

Por otro lado debemos calcular la cantidad de objetos reservados producto de invocaciones a otros métodos. Para esto se utiliza la especificación del método llamado y se calcula la suma del estimador paramétrico $\mathcal{O}(\mathcal{I}^m_{lis}, objectsAlloc(m'), P_m)$ para todo $lis$ pertenciente a $LIS_m$.

Para un método $m$, el algoritmo 1 calcula de manera composicional el estimador paramétrico $objectsAlloc(m)$. Notar que en este esquema un método no analizable (por ejemplo, porque es complejo determinar apropiadamente los invariantes) no representa un obstáculo para el análisis. El programador podría enriquecer el análisis especificando la cantidad de objetos que el método en cuestión reserva permitiendo resolver el cálculo en su completitud.

**Algoritmo 1** Cantidad de objetos reservados por $m$

```
total ← 0
for lcs ∈ LCS_m do
  total ← total + O(I^m_lcs, P_m)
end for
for lis = m.l.m' ∈ LIS_m do
  total ← total + O(I^m_lis, objectsAlloc(m'), P_m)
end for
return total
```

**Ejemplo** Consideremos el método $m1$ del ejemplo 3.1. Para $m1$ hay un sitio de creación $LCS_{m1} = \{\ m1.4\ \}$ y un sitio de invocación $LIS_{m1} = \{\ m1.5\ \}$. El método $m2$ reserva $2n+1$ objetos, es decir $objectsAlloc(m2) = 2n+1$. La cantidad de objetos reservados por $m1$ puede determinarse de la siguiente manera:

$$objectsAlloc(m1) = \mathcal{O}(\mathcal{I}^{m1}_{m1.4}, P_{m1}) + \mathcal{O}(\mathcal{I}^{m1}_{m1.5}, objectsAlloc(m2), P_{m1})$$

Considerando el invariante para $m1.4$:

$$\mathcal{I}^{m1}_{m1.4} = \{1 \leq i \leq k\}$$

$$\mathcal{O}(\mathcal{I}^{m1}_{m1.4}, P_{m1}) = k$$

Considerando el invariante para $m1.5$ y la expresión de consumo de $m2$:

$$\mathcal{I}^{m1}_{m1.4} = \{1 \leq i \leq k, i = n\}$$

$$objectsAlloc(m2) = 2n + 1$$

$$\mathcal{O}(\mathcal{I}^{m1}_{m1.5}, objectsAlloc(m2), P_{m1}) = k^2 + 2k$$

Podemos determinar el estimador paramétrico de la cantidad de objetos reservados por $m1$ como:

$$objectsAlloc(m1) = k^2 + 3k$$

□

## 4.2. Analizando programas

El algoritmo 1 utiliza para determinar la cantidad de objetos creados por un método $m$ la especificación de la cantidad de objetos creados por los métodos llamados por $m$. Esto impone una condición sobre el orden en que los métodos son analizados ya que no es posible analizar $m$ sin conocer el efecto que produce la ejecución de los métodos que $m$ invoca. Este problema puede ser resuelto de diferentes formas, el enfoque utilizado en esta tesis es calcular el grafo de llamadas y determinar un orden sobre el mismo considerando las dependencias que las llamadas determinan.

**Definición** Un *ordenamiento topológico* de un grafo acíclico es todo ordenamiento $m_1, \ldots, m_k$ de los nodos del grafo tal que si $m_i \rightarrow m_j$ es un eje del grafo entonces $m_i$ aparece antes que $m_j$ en el ordenamiento. □

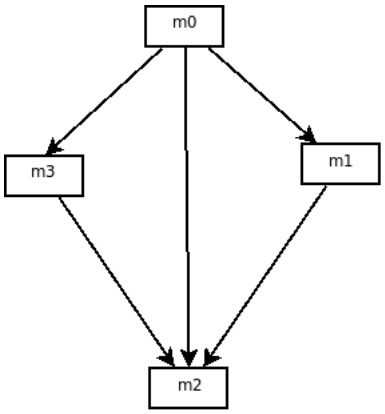


Figura 4.1: Árbol de llamadas.

**Ejemplo** Un ordenamiento topológico para el árbol de llamadas 4.1 correspondiente al ejemplo 3.1 es: $\{m0,\ m3,\ m1,\ m2\}$ □

Considerando el árbol de llamadas $CG$ para un programa $p$, podemos determinar un ordenamiento topológico $\mathcal{OT}$ de $GC$ y aplicar el análisis de requerimientos de memoria siguiendo un orden inverso. El algoritmo 2 analiza los métodos de un programa siguiendo estas condiciones.

**Algoritmo 2** Análisis de objetos reservados por un programa $p$

```
cg ← p.CallGraph
orden ← OT(cg)
orden ← orden.reverse()
for m in orden do
  analizar(m)
end for
```

Notar que el árbol de llamadas $GC$ para el programa $p$ debe ser acíclico. Esta condición implica que, a priori, el análisis no soporta métodos recursivos. Más adelante se discutirá alternativas a esta restricción.

La figura 4.2 muestra para cada iteración del algoritmo 2, siguiendo el orden propuesto anteriormente, el árbol de llamadas del ejemplo 3.1 decorado con la expresión *objectAlloc* calculada en ese paso.

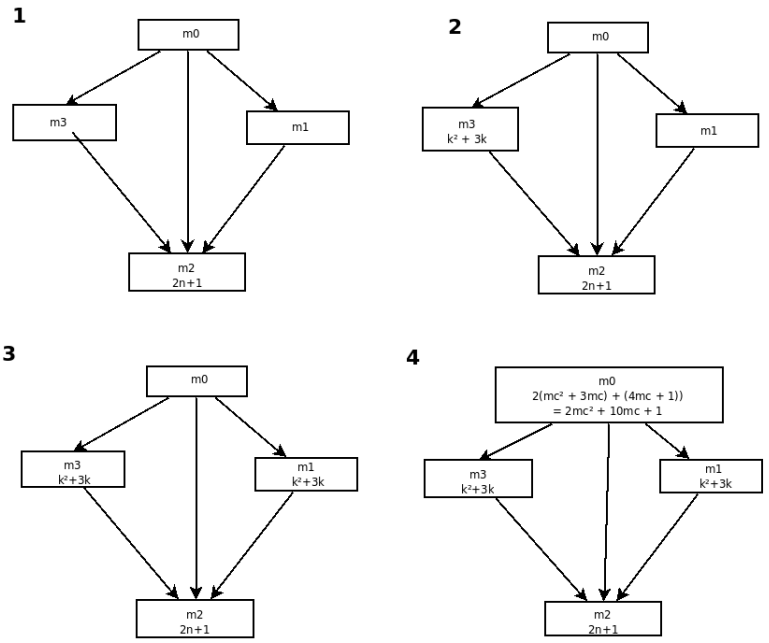


Figura 4.2: Cálculo del estimador paramétrico *objectsAlloc*

## 4.3. Cálculo composicional de requerimientos de memoria

En la sección anterior presentamos una algoritmo para calcular la cantidad total de objetos reservados por la ejecución de un método. Este algoritmo no tiene en cuenta la posibilidad de que el administrador de memoria libere los objetos que no serán nuevamente referenciados.

En un esquema de memoria por *scopes* los objetos son agrupados en regiones de memoria que son asociadas con el tiempo de vida de una unidad computacional. En este esquema, los objetos pueden ser recolectados cuando su unidad de ejecución finaliza.

En particular, vamos a considerar un esquema de administración de memoria que respete las siguientes condiciones:

1. Un objeto creado por un método $m$ puede ser recolectado al finalizar la ejecución de $m$ si su tiempo de vida no es mayor al de $m$.

2. Un objeto creado por la ejecución de un método $m'$ llamado por otro método $m$ puede ser recolectado al finalizar la ejecución de $m$ si su tiempo de vida no es mayor al de $m$.

3. Un objeto creado por un método $m$ no puede ser recolectado al finalizar la ejecución de los métodos llamados por $m$. Es decir, ningún método puede recolectar objetos creados por sus predecesores en el árbol de llamadas.

Notar que el item 3 implica una restricción fuerte sobre el administrador de

memoria ya que los objetos no pueden ser liberados en el mismo momento que dejan de ser referenciados. Un administrador de memoria podría detectar cuando un objeto no es más referenciado, a partir de ese momento el objeto puede ser recolectado.

Para calcular la información de alcance de los objetos se utilizan técnicas de análisis de escape y punteros [SR01, Bla99, SYG05]. La sección 5.3 presenta un esquema de administración de memoria por regiones que satisface las condiciones enunciadas y un algoritmo de análisis estático que permite calcular el tiempo de vida de los objetos.

### 4.3.1. Memoria temporal y residual de un método

Los objetos creados durante la ejecución de un método pueden dinstinguirse en dos conjuntos, los que escapan a $m$ y los que son capturados por $m$. A los objetos que son capturados por $m$ los denominamos temporales, pueden ser recolectados al finalizar la ejecución de $m$ y en su conjunto constituyen el *temporal* de $m$.

**Definición** Sea un método $m$ denominamos $Temporal_m$ al conjunto de objetos creados durante la ejecución de $m$ y los métodos llamados por $m$ que no escapan a $m$.

Los objetos que escapan a $m$ pueden ser vistos como el residuo producido por la ejecución de $m$, ya que no pueden ser recolectados al finalizar la ejecución de $m$. Estos objetos son denominados residuales y en su conjunto forman el *residual* de $m$.

**Definición** Sea un método $m$ denominamos $Residual_m$ al conjunto de objetos creados durante la ejecución de $m$ y los métodos llamados por $m$ que escapan a $m$.

Para determinar el conjunto de objetos temporales y residuales de un método $m$ debemos analizar los objetos creados por $m$ y los objetos creados durante la ejecución de los métodos llamados por $m$.

Analicemos primero el comportamiento de un método sin considerar las llamadas realizadas por éste. Consideremos el conjunto de los objetos creados por $m$, es decir, aquellos objetos que son creados por un sitio de creación $lcs \in LCS_m$.

**Definición** Sea $m$ un método, definimos $TempLocal_m$ como el conjunto de objetos creados por $m$ que no escapan a $m$ y $ResLocal_m$ como el conjunto de objetos creados por $m$ que escapan a $m$.

Para inferir requerimientos de memoria hace falta determinar el cardinal de $TempLocal_m$ y $ResLocal_m$. La técnica utilizada en la sección 4.1.1 permite obtener un estimador paramétrico, en función de los parámetros de $m$, de la cantidad de objetos creados por los puntos de creación de $m$. Si el objeto creado por el sitio de creación no escapa al método entonces es considerado parte del *temporal*, caso contrario es considerado residual.

**Definición** Sea $m$ un método y $lcs$ un punto de creación de $m$, la función $Escapa(m, lcs)$ determina si el objeto creado por $lcs$ escapa a $m$[1].

**Ejemplo** Para $m1$ el sitio de creación $m1.4$ no escapa a $m1$ mientras que el sitio de creación $m2.8$ escapa de $m2$. □

Utilizando la cota superior de la cantidad de objetos creados por un punto de creación $lcs$ determinado por el estimador $\mathcal{O}(\mathcal{I}^m_{lcs}, P_m)$ y la función $Escapa(m, lcs)$, el algoritmo 3 aproxima el cardinal de los conjuntos $TempLocal_m$ y $ResLocal_m$.

**Algoritmo 3** Cálculo de $TempLocal_m$ y $ResLocal_m$.

```
residual ← 0
temporal ← 0
for lcs ∈ LCS_m do
  if Escapa(m, lcs) then
    residual ← residual + O(I^m_lcs, P_m)
  else
    temporal ← temporal + O(I^m_lcs, P_m)
  end if
end for
return (residual, temporal)
```

Consideremos ahora los objetos residuales producto de las llamadas realizadas por el método $m$. Por cada llamada de $m$ a un método $m'$ debemos considerar el *residual* de $m'$. Los objetos creados durante la ejecución de $m'$ pueden escapar o no al contexto de $m$, esto determina si deben ser tratados como parte del residual o temporal de $m$.

**Definición** Sea $m$ un método, definimos $TempCall_m$ como el conjunto de objetos residuales de los métodos llamados por $m$ que no escapan a $m$ y $ResCall_m$ como el conjunto de objetos residuales de los métodos llamados por $m$ que escapan a $m$. Más precisamente:

$$TempCall_m = \bigcup_{m.l.m_i \in \mathcal{LIS}_m} \{\ x \mid x \in Residual_{m_i} \wedge \neg escapa(m, x)\ \}$$

$$ResCall_m = \bigcup_{m.l.m_i \in \mathcal{LIS}_m} \{\ x \mid x \in Residual_{m_i} \wedge escapa(m, x)\ \}$$

□

[1] La sección 5.3.2 presenta una técnica para el cálculo de la función $Escapa(m, lcs)$

Para inferir requerimientos de memoria, debemos calcular el cardinal de estos conjuntos:

$$\#TempCall_m = \sum_{m.l.m_i \in \mathcal{IS}_m} \#\{\ x \mid x\ \in Residual_{m_i} \wedge\ \neg escapa(m,x)\ \}$$

$$\#ResCall_m = \sum_{m.l.m_i \in \mathcal{IS}_m} \#\{\ x \mid x\ \in Residual_{m_i} \wedge\ escapa(m,x)\ \}$$

Cuantificar estos conjuntos es un problema complejo. Esto se debe a que no alcanza con conocer el cardinal del residual de un método $m'$ que es invocado, sino que es necesario conocer información sobre las referencias a los objetos producidas por la ejecución del método $m'$, para poder determinar luego cómo evolucionan estas referencias en el método llamador. En este trabajo, esta problemática es resuelta especificando el residual de un método teniendo en cuenta el resultado de un análisis particular de escape [SYG05] que será presentado en la sección 5.3. Utilizando este enfoque, la información brindada por este análisis es enriquecida con información de consumo (ver sección 5.4).

Dado un sitio de invocación $lis = m.l.m'$ de $m$, llamaremos $C^m_{lis}$ y $R^m_{lis}$ a expresiones en función de los parámetros de $m'$ que determinan la cantidad de objetos residuales de $m'$ que son capturados y que escapan a $m$.

**Ejemplo** Para $m3.\,13.m2$ el arreglo retornado por $m2$ es a su vez, retornado por $m3$. Luego estos objetos constituyen el residual de $m3$ siendo $R^{m1}_{m3.13.m2} = 2n + 1$. Por otro lado, el análisis de escape utilizado determina que todos los objetos de $m2$ escapan y son capturados por $m1$ en el sitio de invocación $m1.\,5.m2$ luego $C^{m1}_{m1.5.m2} = 2n + 1$. □

Dado $lis = m.l.m'$ un punto de invocación de $m$, $\mathcal{I}^m_{is}$ el invariante asociado a $lis$ y las expresiones $C^m_{is}$ y $R^m_{is}$ utilizando el estimador presentado en la sección 4.1.2 podemos calcular $\#TempCall_m$ y $\#ResCall_m$ de la siguiente manera:

$$\begin{aligned}
\#TempCall_m &= \sum_{is \in LIS_m} \mathcal{O}(\mathcal{I}^m_{lis}, \mathcal{C}^m_{lis}, P_m) \\
&= \sum_{is \in LIS_m} \mathcal{CP}(\mathcal{I}^m_{lis}, \mathcal{C}^m_{lis}, P_m) \\
\#ResCall_m &= \sum_{is \in LIS_m} \mathcal{O}(\mathcal{I}^m_{lis}, \mathcal{R}^m_{lis}, P_m) \\
&= \sum_{is \in LIS_m} \mathcal{CP}(\mathcal{I}^m_{lis}, \mathcal{R}^m_{lis}, P_m)
\end{aligned}$$

**Ejemplo** Para el ejemplo 3.1 $LIS_{m1} = \{\ m1.5.m2\ \}$, $C^m_{m1.5.m2}1 = 2n+1$ y $\mathcal{I}^{m1}_{m1.5} = \{1 \leq i \leq k,\ i = n\}$ luego $\#TempCall_{m1}$ queda determinado por $\mathcal{O}(\{1 \leq i \leq k,\ i = n\}, 2n+1, k) = k^2 + 2k$. Siendo que el residual de $m2$ es capturado por $m1$ $R^m_{m1.5.m2}1 = 0$.

Por otro lado, para $m3$ $LIS_{m3} = \{\ m3.13.m2\ \}$, $R^{m3}_{m3.13.m2} = 2n+1$ y $\mathcal{I}^{m3}_{m3.13.m2} = \{\ 1 \leq j \leq k,\ n = j\ \}$ entonces $\#ResCall_{m3}$ queda determinado por $\mathcal{O}(\{\ 1 \leq j \leq k,\ n = j\ \}, 2n+1, k) = k^2 + 2k$. Siendo que todos los objetos que escapan a $m2$ a su vez escapan a $m3$ $C^{m3}_{m3.13.m2} = 0$. □

El algoritmo 4 calcula $\#TempCall_m$ y $\#ResCall_m$. Notar que para el cálculo asumimos que las expresiones $\mathcal{C}^m_{lis}$ y $\mathcal{R}^m_{lis}$ son conocidas. En la sección 5.3.2 presentamos como se calculan estas expresiones.

**Algoritmo 4** Cálculo de $TempCall_m$ y $ResCall_m$.

```
residual ← 0
temporal ← 0
for lis ∈ LIS_m do
    temporal ← temporal + O(I^m_lis, C^m_lis, P_m)
    residual ← residual + O(I^m_lis, R^m_lis, P_m)
end for
return (residual, temporal)
```

### 4.3.2. Analizando la memoria temporal

Considerando la recolección de objetos en el esquema de administración de memoria asumido podemos definir, para un método $m$, un nuevo estimador paramétrico $liveObjects_m$ que determina el mayor número de objetos vivos reservados durante la ejecución de $m$ y todos los métodos invocados por $m$. Teniendo en cuenta el tamaño de los objetos, este estimador es equivalente a la expresión $memReq_m$ definida en la sección 3.2.1. Para calcular $liveObjects_m$ debemos considerar:

1. Los objetos creados por $m$ que pueden escapar o no a $m$.
2. Los objetos residuales de los métodos llamados por $m$ que pueden ser capturados por $m$ o que a su vez escapan a $m$.
3. Los objetos temporales de los métodos llamados por $m$ que son liberados al finalizar la ejecución de $m$.

El item 1 fue tratado anteriormente y corresponde al cálculo de las expresiones $\#TempLocal_m$ y $\#ResLocal_m$ (ver algoritmo 3).

El item 2 fue tratado en la sección anterior y corresponde al cálculo de las expresiones $\#TempCall_m$ y $\#ResCall_m$ (ver algoritmo 4).

El item 3 requiere analizar la memoria temporal de los métodos invocados por el método bajo análisis. Consideremos por ejemplo el método $m0$ el cual realiza dos llamadas en los puntos $m0.1.m1$ y $m0.2.m2$. Al finalizar la ejecución de $m1$ y $m2$ los objetos temporales de $m1$ y $m2$ pueden ser recolectados por el administrador de memoria (ver figura 3.1).

Dado un método $m$, para determinar $liveObjects_m$ debemos calcular cuál de las llamadas realizadas por $m$ reserva el mayor número de objetos temporales. Es decir, para todo punto de invocación $lis \in LIS_m$ debemos calcular un estimador paramétrico de la cantidad de objetos temporales reservados por $lis$ y seleccionar aquel que maximice los requerimientos de memoria de $m$. Notar que para un punto de invocación, el temporal depende de los párametros con los que se realiza la llamada. Por ejemplo, $m3$ llama a $m2$, cuyo temporal es $2n + 1$, en un ciclo con valores crecientes de $n$ en cada iteración. Luego, el temporal de $m2$ adopta el mayor tamaño cuando $m2$ es invocado con $n = k$. La figura 4.3 muestra el crecimiento del temporal de $m2$ en cada iteración.

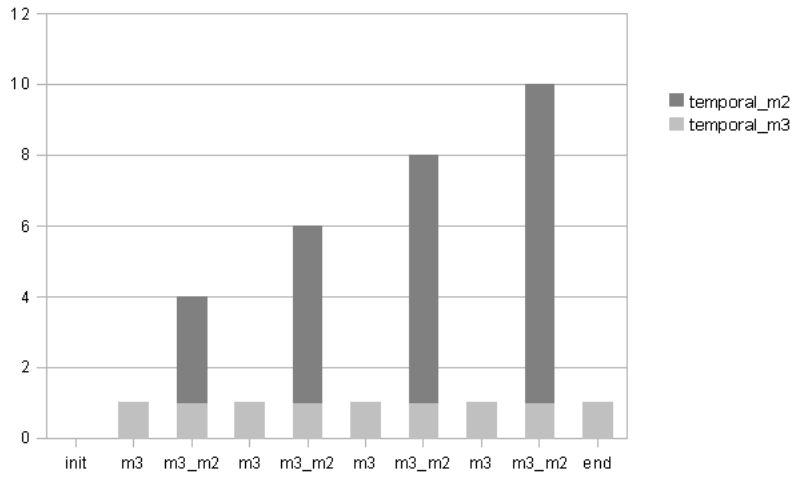


Figura 4.3: Comportamiento del temporal de $m2$

Sea $lis = m.l.m'$ un punto de invocación para un método $m$ de parámetros $P_m$, $lis.target = m'$, $\mathcal{I}^m_{lis}$ el invariante asociado a $lis$ y $Temporal_{m'}$ una expresión que acota la cantidad de objetos temporales de $m'$. $MaxInv(is, \mathcal{I}^m_{lis}, P_m)$ determina una expresión paramétrica que maximiza la cantidad de objetos temporales reservados por $lis$ sujeto al invariante $\mathcal{I}^m_{lis}$.

$$MaxInv(is, \mathcal{I}^m_{lis}, P_m) = \mathcal{M}(\mathcal{I}^m_{lis}, Temporal_{lis.taget}, P_m)$$

La caracterización de este estimador determina un problema de maximización no lineal (ver sección 2.1.2) cuya solución es una expresión paramétrica en función de $P_m$.

**Ejemplo** Para el punto $m1.\,5.m2$ el máximo número de objetos temporales queda determinado por :

$$MaxInv(m0.\,1.m1, \mathcal{I}^{m0}_{m0.1.m1}, P_{m0})$$
$$= \mathcal{M}(\{0 \leq i \leq k,\ i\ =\ n\}, 2n+1, \{k\})$$
$$= 2k+1$$

□

Como mencionamos anteriormente, debemos calcular de todas las llamadas realizadas por $m$, cuál de estas maximiza la cantidad de objetos reservados.

**Definición** Sea $m$ un método y el conjunto de puntos de creación $IS_m$ de $m$, definimos $MaxCall_m$ como:

$$MaxCall_m = \text{máx}\{\ MaxInv(lis, \mathcal{I}^m_{lis}, P_m) \mid lis\ in\ LIS_m\}$$

Al tener en cuenta la recolección de objetos redefinimos el temporal de un método $m$ considerando de todas las llamadas que $m$ realiza, el mayor de los temporales, es decir, la expresión $MaxCall_m$.

**Definición** Sea un método $m$, definimos el estimador paramétrico $\#Temporal_m$ como:

$$\#Temporal_m = \#TempLocal_m + \#TempCall_m + MaxCall_m$$

**Ejemplo** Para calcular $\#Temporal_{m0}$ debemos tener en cuenta las llamadas que realiza $m0$. Como mencionamos antes los objetos creados durante la ejecución de $m1$ son temporales $(k^2+3k)$, luego el temporal de $m1$ al ser llamado por $m0$ con $k = mc$ es de $mc^2 + 3mc$. Por otro lado $m2$ no tiene temporal, pero genera un residual de $2n+1$ objetos que son capturados por $m0$. Como $m0$ invoca a $m2$ con $n = 2mc$ los objetos capturados por $m0$ son $4mc+1$. A su vez, $m3$ genera un residual de $k^2+2k$

objetos capturado por $m0$ y un temporal de $k$ objetos. Como $m0$ invoca a $m3$ con $k = mc$ entonces $m0$ captura $mc^2 + 2mc$ objetos y el temporal es de $mc$.

$$\#Temporal_{m0} = \#TempLocal_{m0} + \#TempCall_{m0} + MaxCall_{m0}$$

$$\#TempLocal_{m0} = 0$$

$$\begin{aligned} \#TempCall_{m0} &= (4mc + 1) + (mc^2 + 2mc) \\ &= mc^2 + 6mc + 1 \end{aligned}$$

$$\begin{aligned} MaxCall_{m0} &= max\{mc^2 + 3mc, 0\ , mc\ \} \\ &= mc^2 + 3mc \end{aligned}$$

Finalmente el temporal de $m0$ queda caracterizado por:

$$\begin{aligned} \#Temporal_{m0} &= mc^2 + 6mc + 1 + mc^2 + 3mc \\ &= 2mc^2 + 9mc + 1 \end{aligned}$$

□

Habiendo determinado una expresión paramétrica para cada uno de los requerimientos enunciados al comienzo de esta sección, podemos definir el estimador paramétrico $liveObjects_m$.

**Definición** Sea un método $m$ el estimador paramétrico $liveObjects_m$ determina una cota superior de la mayor cantidad de objetos vivos para cualquier ejecución de $m$.

$$\begin{aligned} liveObjects_m &= \#Temporal_m + \#Residual_m \\ &= \#TempLocal_m + \#TempCall_m + MaxCall_m \\ &\quad + \#ResLocal_m + \#ResCall_m \end{aligned}$$

□

Finalmente, el algoritmo 4.3.2 presenta el cálculo del estimador paramétrico $liveObjects_m$. Notar que para este cálculo el algoritmo calcula todas las expresiones asociadas al temporal y residual de un método.

**Ejemplo** El método $m0$ no genera residual, luego para determinar $liveObjects_{m0}$ basta considerar el temporal que calculamos anteriormente. Notar que la expresión resultante difiere del estimador $objectsAlloc_{m0}$ en $mc$ objetos. Esto se debe a que los

objetos creados en $m3.14$ son capturados por $m3$ y al considerar los temporales de las llamadas que realiza $m0$ determinamos que el máximo temporal es generado por $m1$.

$$\begin{aligned} liveObjects_{m0} &= \#Temporal_{m0} \\ &= 2mc^2 + 9mc + 1 \end{aligned}$$

□

**Algoritmo 5** Algoritmo completo para el cálculo de $liveObjects_m$

$residualLocal,\ temporalLocal \leftarrow 0$
$residualCall,\ temporalCall \leftarrow 0$
$maxCall \leftarrow 0$
**for** $lcs\ in\ LCS_m$ **do**
  $cantidad \leftarrow \mathcal{O}(\mathcal{I}^m_{lcs}, P_m)$
  **if** $Escapa(m, lcs)$ **then**
    $residualLocal \leftarrow residualLocal + cantidad$
  **else**
    $temporalLocal \leftarrow temporalLocal + cantidad$
  **end if**
**end for**
**for** $lis\ in\ LIS_m$ **do**
  $temporalCall \leftarrow temporalCall + \mathcal{O}(\mathcal{I}^m_{lis}, \mathcal{C}^m_{lis}, P_m)$
  $residualCall \leftarrow residualCall + \mathcal{O}(\mathcal{I}^m_{lis}, \mathcal{R}^m_{lis}, P_m)$
  $maxCall \leftarrow max(maxCall, (MaxInv(is, \mathcal{I}^m_{lis}, P_m))$
**end for**
$temporal \leftarrow temporalLocal + temporalCall + maxCall$
$residual \leftarrow residualLocal + residualCall$
$liveObjects_m \leftarrow temporal + residual$

## 4.4. Conclusiones

En este capítulo presentamos un análisis composicional para determinar una cota paramétrica de la cantidad total de objetos reservados por un método (el algoritmo 1 determina esta cota).

Luego, al considerar la memoria que puede ser liberada durante la ejecución de un método, presentamos un análisis composicional para calcular la expresión *liveObjects* que determina el máximo número de objetos vivos durante la ejecución de un método $m$ (el algoritmo 4.3.2 calcula esta expresión).

A diferencia de [BGY06, BFGY08] los puntos de creación no son modelados como un camino en el árbol de llamadas, sino que se consideran los puntos de creación locales al método bajo análisis. Para tener en cuenta el efecto producido por las llamadas a métodos se definen operaciones de suma y maximización en función del invariante de la llamada que permiten acumular el residual y maximizar el temporal de los métodos invocados.

Este nuevo enfoque permite que los invariantes utilizados sean locales y no globales, lo que es deseable ya que los invariantes globales son en general complejos y difíciles de manipular.

A su vez, el cálculo composicional permite incorporar al análisis métodos no analizables por la técnica propuesta siempre y cuando el comportamiento de dicho método sea especificado. Esta especificación podría ser conocida de antemano, generada por una herramienta externa o bien dada por el programador.

Para implementar la técnica de análisis composicional descripta a lo largo de este capítulo es necesario:

1. Contar soluciones enteras de un invariante.

2. Maximizar y sumar expresiones paramétricas en función de un invariante.

3. Calcular la función $Escapa(lcs, m)$ y las expresiones $R^m_{lis}$ y $C^m_{lis}$.

El siguiente capítulo presenta las técnicas seleccionadas que permiten resolver cada uno de estos problemas.

# Capítulo 5

# Calculando efectivamente requerimientos de memoria

En el capítulo anterior presentamos un análisis composicional para el cálculo de requerimientos de memoria. Dado un método $m$ con parámetros $P_m$ se obtiene una cota superior paramétrica ($objectsAlloc$) de la cantidad de objetos reservados por la ejecución de $m$ . Luego, al considerar la liberación de memoria definimos un estimador en función de $P_m$ ($liveObjects$) que determina el mayor número de objetos vivos durante la ejecución de $m$.

Para calcular estos estimadores contamos el número de soluciones enteras para un invariante (ver sección 2.1.1), el número de soluciones enteras de un invariante asociando un peso a cada solución (ver sección 2.1.1) y maximizamos expresiones paramétricas sobre un dominio determinado (ver sección 2.1.2). Para estas operaciones los invariantes utilizados determinan el espacio de iteración de los sitios de creación e invocación. En el caso de llamadas a métodos, los invariantes vinculan las variables locales del método llamador con los parámetros del método llamado.

Para el cálculo composicional del estimador $liveObjects$ asumimos que, para un sitio de invocación $is = m.l.m'$, podemos determinar el cardinal del conjunto de objetos pertenecientes al residual de $m'$ que son capturados por $m$ y aquellos que a su vez escapan a $m$. Estos estimadores fueron caracterizados por las expresiones $C^m_{lis}$ y $R^m_{lis}$.

La implementación de una herramienta que permita calcular de forma composicional requerimientos de memoria requiere la solución de cada uno de los problemas enunciados. En este capítulo presentamos las decisiones de diseño tomadas para el desarrollo de un prototipo que implemente el cálculo composicional presentado en el capítulo anterior. A su vez, presentamos las restricciones que imponen las técnicas elegidas para la resolución de los distintos problemas.

## 5.1. Generación de invariantes

Las técnicas utilizadas para calcular el uso de memoria dependen de invariantes que restringen las valuaciones posibles de las variables para un punto determinado del programa y en el caso de llamadas a métodos los invariantes no sólo restringen las valuaciones de las variables sino que determinan la relación entre las variables locales del método llamador y los parámetros del método llamado.

Los invariantes pueden ser provistos por el programador o calculados usando técnicas de análisis estático. En este trabajo ambas alternativas son exploradas.

La herramienta desarrollada para el cálculo composicional permite especificar manualmente los invariantes requeridos. Esto puede ser realizado mediante una *api* provista por la herramienta o mediante la especificación de una archivo con formato xml.

La posibilidad de especificar los invariantes requeridos mediante un archivo define un potencial punto de integración con herramientas de generaración automática de invariantes. La integración puede ser resuelta traduciendo las salidas de dichas herramienta al formato requerido o bien realizando una extensión de la herramienta que genera los invariantes para obtener el formato esperado como salida.

Para generar invariantes de forma automática nuestro prototipo utiliza una herramienta desarrollada por Garbervetsky [Gar05]. La integración fue resuelta traduciendo la información obtenida por dicha herramienta al formato xml requerido por nuestro prototipo (ver sección 6.1).

En [Gar05] se utiliza Daikon como base para la implementación del análisis. Básicamente, la técnica genera nuevas variables para expresiones que asume tendrán efecto en el número de veces que un sitio del programa es visitado y produce un método *dummy* antes de cada punto de interés cuyos argumentos son las variables detectadas. Usando este procedimiento la precondición del método generado contiene un invariante para el punto instrumentado del programa que predica únicamente sobre las variables especificadas.

## 5.2. Cuantificando el uso de memoria

Las técnicas presentadas para determinar el uso de memoria dependen de la habilidad de contar el número de elementos que satisfacen un invariante. Si el invariante puede ser representado por un sistema de restricciones lineales, entonces estos problemas son equivalentes a contar el número de puntos enteros en un politopo paramétrico (ver sección 2.1). Para sintetizar los estimadores paramétricos del uso de memoria, además del problema de aproximar el número de visitas de un punto de creación, debemos resolver la máximización de una expresión paramétrica sobre un conjunto de restricciones 2.1.2 y la suma de evaluar una expresión paramétrica

sobre todos los puntos que satisfacen un conjunto de restricciones 2.1.3 (equivalente a contar cantidad de soluciones asociando a cada una un peso).

Para la resolución de estos problemas utilizamos las técnicas implementadas en Barvinok [VSB+04] mediante la integración de la biblioteca *libbarvinok* [Ver07]. En las siguientes secciones se detallan las operaciones utilizadas, los detalles de los algoritmos implementados en Barvinok exceden el alcance de esta tesis por lo cual utilizaremos *libbarvinok* como una caja negra. La integración de esta librería a nuestro prototipo es presentada en la sección 6.3.

### 5.2.1. Enumerando conjuntos de restricciones

Este problema fue enunciado en la sección 2.1.1 y consiste en la enumeración de los puntos enteros que satisfacen un conjunto de restricciones. Para contar el número de soluciones de un predicado debemos definir que variables son libres y cuáles no.

**Ejemplo** Consideremos el siguiente invariante:

$$\mathcal{I} = \{k = mc, 1 \leq i \leq k, n = i,\ 1 \leq j \leq n\}$$

Sea $mc$ la única variable libre, es decir, un parámetro. Entonces el número de soluciones de $\mathcal{I}$ en función de $mc$ es:

$$\begin{aligned} \mathcal{C}(\mathcal{I}, \{mc\}) &= \#\{(k,i,j,n) | k = mc, 1 \leq i \leq k, n = i,\ 1 \leq j \leq n\} \\ &= \frac{1}{2}(mc^2 + mc) \end{aligned}$$

Cuando las restricciones son lineales, puede utilizarse el modelo de politopos paramétricos 2.1 para su resolución. Clauss y Loechner[CL98] demostraron que el número de puntos enteros de un politopo paramétrico $P_p$ puede ser representado por un con conjunto de quasi-polinomios cada uno asociado a una partición del espacio de los parámetros $p$.

### 5.2.2. Maximizando expresiones paramétricas

Como mencionamos anteriormente, para calcular la memoria requerida por un método bajo un modelo de memoria por regiones, debemos calcular el máximo de una expresión paramétrica restringido a un conjunto de restricciones. Mas aún, el conjunto de restricciones es utilizado para asociar las variables locales y parámetros del método bajo análisis con los parámetros de la expresión paramétrica a maximizar. En particular, si las restricciones son lineales, es posible representar el dominio utilizando el modelo de politopos paramétricos [Fea96].

Considerando que las expresiones paramétricas a evaluar son básicamente el resultado de sumar expresiones obtenidas por la operación de enumeración descripta

en la sección anterior, entonces el problema de maximización es reducido a la maximización de polinomios sobre todos los puntos enteros en un politopo paramétrico, resultando una expresión que depende solamente en los parámetros estructurales.

Para resolver este problema utilizamos el enfoque presentado en [CFGV09] mediante la implementación definida en [Ver07]. Dada una expresión paramétrica $\mathcal{F}$ definida por un conjunto de polinomios, asociado cada uno a una partición del espacio de parámetros $p$ y un conjunto de restricciones lineales que define un politopo paramétrico $P_p$, la técnica obtiene un conjunto de polinomios candidatos que acotan la expresión $\mathcal{F}$ en el dominio determinado por $P_p$

**Ejemplo** Maximización del polinomio $\mathcal{F}$ sobre el dominio $\mathcal{I}$,

$$\mathcal{F}(n,m) = n^2 + 2m$$

$$\begin{aligned} \mathcal{M}(\mathcal{I}, \mathcal{F}(n,m), \{P,Q\}) &= \text{máx}\{n^2 + 2m |\ i = n, m = 4Q, 1 \leq n, n \leq P, 0 \leq Q\} \\ &= 8Q + P^2 \end{aligned}$$

□

Notar que para este ejemplo, la técnica implementada en [Ver07] retorna un sólo candidato. Sin embargo, hay casos en donde no puede determinar un único polinomio que acote la expresión $\mathcal{F}$.

**Ejemplo** Para el polinomio $\mathcal{F}$ y el invariante $\mathcal{I}$ la herramienta genera más de un candidato:

$$\mathcal{F}(n) = n^2 - 1$$

$$\mathcal{M}(\mathcal{I}, \mathcal{F}(n), \{P,Q\}) = \text{máx}\{n^2 - 1 | 1 \leq i \leq P + Q, i \leq 3Q, n = i\}$$

$$\begin{cases} \{(P+Q)^2, P + Q - 1\} & \text{si } 2Q \geq P \\ 9P^2 - 1 & \text{si } 2Q \leq P. \end{cases}$$

□

### 5.2.3. Suma de una expresión paramétrica sobre un dominio

Para calcular la cantidad de objetos reservados por un método considerando la especificación de los métodos llamados, es necesario poder sumar expresiones paramétricas sobre un dominio (invariante). Este dominio determina el espacio de

iteración del punto del programa en el que se realiza la llamada y a su vez vincula las variables y parámetros del método llamador con el método llamado.

En [VB08] se presentan diferentes métodos para el cálculo de este problema, cuando el dominio puede ser representado por un politopo paramétrico y la función de costo es polinomial. Estos métodos son implementados en la herramienta [Ver07].

**Ejemplo** Suma obtenida por [Ver07] para $\mathcal{F}$ sobre el dominio $\mathcal{I}$:

$$\mathcal{F}(n) = n^2 - 1$$

$$\begin{aligned} \mathcal{S}(\mathcal{I}, \mathcal{F}(n), \{P, Q\}) &= \sum_{1 \leq i \leq P+Q, i \leq 3Q, n=i} n^2 - 1 \\ &= P^3 \end{aligned}$$

□

Es importante notar que el resutado obtenido es una expresión en función de los parámetros del politopo, es decir, de los parámetros del método bajo análisis.

### 5.2.4. Limitaciones

Una limitación que imponen las técnicas presentadas anteriormente es que los invariantes deben ser lineales y numéricos. Esto restringe el espacio de programas sobre el cual podemos aplicar el análisis composicional.

Por otro lado, tanto la operación de suma como la de máximo de una expresión paramétrica sobre un dominio se restringen a expresiones polinomiales. Si bien las expresiones que consideramos para sumar y maximizar son esencialmente el resultado de enumerar invariantes lineales lo que determina expresiones polinomiales el análisis composicional permitiría especificar expresiones no polinomiales para un método no analizable (por ejemplo, porque los invariantes no son lineales). Sin embargo, no podríamos continuar manipulando estas expresiones.

Determinar el máximo entre un conjunto de polinomios de manera simbólica es un problema que no ha sido resuelto completamente. Como vimos anteriormente la operación de maximización puede sugerir más de un candidato. A su vez, para determinar el temporal de las llamadas que maximiza los requerimientos de memoria, debemos elegir nuevamente el máximo de un conjunto de polinomios. Existen algunas aproximaciones a este problema sin embargo continúa siendo un problema abierto.

Como el análisis composicional utiliza las expresiones para especificar el resumen de un método y este resumen es utilizado nuevamente para inferir el resumen de los métodos que lo invocan puede ocurrir que haya que resolver operaciones con la forma $maxInv(max(e1, e2))$. Estas expresiones son complejas de evaluar, sobre todo si los candidatos al máximo son más de dos. Distribuir la operación de $maxInv$ obteniendo $max(maxInv(e1), maxInv(e2))$ es una alternvativa, sin embargo esta operación

podría determinar nuevamente múltiples candidatos para el máximo, complejizando más aún la expresión inicial.

En nuestro prototipo, cuando no podemos determinar el máximo utilizando *libbarvinok* utilizamos una aproximación que considera la suma de los polinomios. Esta aproximación puede determinar expresiones demasiados pesimistas. Mejoras en este aspecto constituyen una línea de trabajo futuro.

## 5.3. Análisis de escape

Un esquema de administración de memoria por regiones (ver sección 2.3.1) satisface las condiciones impuestas sobre el administrador de memoria enunciadas en el capítulo anterior. En particular vamos a considerar que, en cada invocación a un método una nueva región es creada la cuál alojará todos los objetos capturados por el método. Al finalizar la ejecución de un método, el administrador de memoria recolecta la región asociada al método y todos los objetos alojados en ésta. Una implementación de memoria por *scopes* siguiendo este enfoque es presentada en [GNYZ04].

La inferencia de regiones requiere determinar los tiempos de vida de los objetos creados dinámicamente por un programa, para esto recurrimos a una técnica de análisis de escape. El prototipo implementado está conectado con el algoritmo de síntesis de regiones propuesto en [SYG05]. Este algoritmo utiliza la hipótesis generacional [JL96] que establece una relación inversa entre la edad de los objetos y su mortalidad. Acorde a esto, se propone ubicar cada estructura de datos en una región distinta. La idea es que la mayoría de los objetos son, o bien de vida corta, entonces pueden ser ubicados en una región de vida corta, o de vida larga, porque están integrados a una estructura mayor y deben ser ubicados de forma conjunta con el resto de la estructura. Por esta razón el análisis presentado no está diseñado para determinar los tiempos de vida absolutos, sino la relación entre los tiempo de vida de los objetos con el fin de predecir qué objetos pertenecen a la misma estructura de datos.

### 5.3.1. Algoritmo de interferencia de punteros

Para cada método $m$, el análisis construye una partición $\sim_m$ de sus variables locales tal que, dos variables realacionadas $v \sim_m v'$ apuntarán a objetos en la misma región. El algoritmo, llamado análisis de interferencia de punteros, trabaja en dos fases.

Durante una primera fase *intra-procedural*, busca todas las variables que interfieren sintácticamente y las marca como parte de la misma clase de equivalencia: $v = u$ , $v = u, f$ o $v.f = u \Rightarrow v \sim_m u$.

En una segunda fase, *inter-procedural*, la interferencia de punteros es modelada usando el árbol de llamadas estático de la siguiente manera: cuando un método

$m$ invoca a un método $m'$ con argumentos $p_1 \leftarrow v_1, \ldots, p_2 \leftarrow v_2, \ldots$ el algoritmo asegura que $p_1 \sim_{m'} p_2$ en $m'$ implica $v_1 \sim_m v_2$ en $m$.

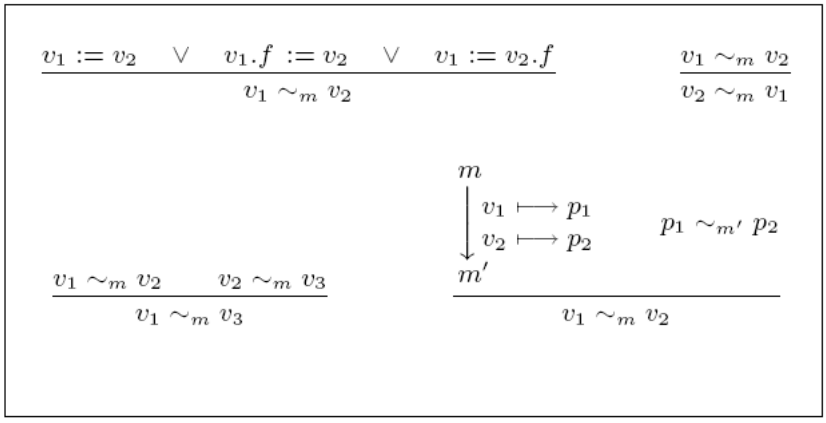


Figura 5.1: El algoritmo de interferencia de punteros

El algoritmo puede ser resumido, como muestra la figura 5.1, como el cálculo de punto fijo del sistema de restricciones dado.

### 5.3.2. Calculando la función $Escapa(m, lcs)$

Para determinar el temporal y residual de un método, en la sección 4.3.1 utilizamos la función $Escapa(m, lcs)$ que determina si el objeto creado por el punto de creación $lcs$ escapa o no al método $m$. Esta función es definida utilizando el resultado del análisis de escape presentado anteriormente.

El análisis determina una partición de las variables de $m$ determinando clases de equivalencia llamadas *familias*. Una familia es una abstracción que modela el hecho de que todos sus miembros pertenecen a la misma estructura de datos y en consecuencia son ubicados en la misma región. Podemos decir intuitivamente que una familia escapa a un método si contiene un parámetro (los parámetros siempre exceden el tiempo de vida del método) o si algún miembro de la familia es retornado (por ejemplo la variable $arrB$ del método m2 es retornada).

**Definición** Una familia escapa a un método $m$ si alguno de sus miembros es retornado al finalizar la ejecución de $m$ o es un parámetro.

Las instrucciones de creación de objetos tienen la siguiente forma: $lcs : v = newClass$, donde $lcs$ es la posición donde se ejecuta la instrucción *new*. Dado que el análisis utiliza una una representación SSA [BP98] del método $m$, basta con tomar $v$ como el representante de $lcs$. De esta forma, la verificación de si el punto de creación local $lcs$ escapa se reduce a mirar a que familia pertenece la variable $v$.

**Definición** Un sitio de creación $lcs : v = newClass$ escapa a un método $m$ si la familia de $v$ escapa a $m$.

Finalmente, la función $Escapa(m, lcs)$ es calculada obteniendo la familia asociada a $lcs$ y verificando si la familia escapa a $m$ según las condiciones enunciadas anteriormente.

## 5.4. Especificación del residual

Como vimos en el capítulo anterior, para analizar el efecto que tiene la invocación a un método, desde el punto del consumo de memoria, es necesario modelar dos tipos de consumo.

Para un método $m$, describimos el consumo temporal de $m$ formado por todos los objetos que son creados durante su ejecución y que son capturados por $m$. Los algoritmos presentados para inferir requeremientos de memoria sólo utilizan el tamaño del temporal ya que al finalizar la ejecución de $m$ estos objetos pueden ser recolectados, consecuentemente no es necesario estudiar su evolución. Por ejemplo, cuando analizamos la llamada que $m0$ realiza a $m1$, siendo que todos los objetos creados por cualquier ejecución de $m1$ son temporales, consideramos el máximo que el temporal de $m1$ puede alcanzar y lo utilizamos para calcular el temporal de $m0$.

Por otro lado, la invocación a un método $m$ puede producir un residuo sobre el método llamador, es decir, del conjunto de objetos creados por la ejecución de $m$, algunos pueden escapar a $m$ y vivir en el contexto del llamador. Por ejemplo, la llamada que $m0$ realiza a $m2$ genera un residuo de $4mc+2$ objetos en $m0$ ya que los objetos creados en $m2$ son retornados.

En la sección 4.3.1 asumimos dos expresiones $C^m_{lis}$ y $R^m_{lis}$ que describen el residual capturado y el residual que escapa respectivamente. Estas expresiones determinan, para un punto de invocación $lis = m.l.m'$ la cantidad de objetos residuales de $m'$ que son capturados por $m$ y la cantidad de objetos residuales de $m'$ que a su vez escapan al contexto de ejecución de $m$. El cálculo de estas expresiones requiere conocer la evolución de los objetos residuales de $m'$ en $m$, es decir, debemos analizar si los objetos residuales son capturados por $m$ o por ejemplo son retornados al finalizar la ejecución de $m$. En el caso de $m3$ el método retorna el arreglo creado en la llamada a $m2$.

La especificación del residual de un método como una expresión paramétrica que especifica cuánto es el residuo generado no es suficiente para seguir la evolución de los objetos residuales en el método llamador. Esto requiere una especificación más completa que involucre la relación entre los objetos residuales. Consideremos nuevamente, a modo de ejemplo, la llamada al método $m2$ en el contexto de $m3$. Si bien, podemos determinar fácilmente que el arreglo retornado por $m2$ a su vez

es retornado por $m3$ no sabemos que pasa con el resto de los objetos residuales de $m2$. Es razonable pensar que todos los objetos que son asignados al arreglo escapen conjuntamente con el arreglo, pero ¿qué pasa con los objetos de tipo C?.

Esta problemática puede ser atacada utilizando técnicas de análisis de escape y punteros. En particular, para inferir el comportamiento de los objetos residuales vamos a utilizar la técnica presentada en la sección anterior.

Retomando el ejemplo, el análisis de escape utilizado determina que todos los objetos de $m2$ forman una familia y escapan de forma conjunta al ser retornado el arreglo. A su vez, el arreglo es retornado en $m3$ en consecuencia inferimos que todos los objetos de la familia escapan a $m3$ componiendo el residual de $m3$.

Enriqueciendo el análisis de escape con expresiones de consumo que cuantifican el tamaño de las familias podemos especificar el residual de un método de manera tal que la evolución de los objetos (agrupados ahora en familias) puede ser inferida en el método llamador. Para un método $m$ de parámetros $P_m$ describimos el residual como:

$$Residual_m = \{\ \{F_1, Size_{F_1}(P_m)\},\ \ldots,\ \{F_n, Size_{F_n}(P_m)\}\ \}$$

donde $F_i$ es una familia que escapa a $m$ y $Size_{F_i}(P_m)$ es una expresión paramétrica en términos de $P_m$ que especifica la cantidad de objetos que forman la familia.

**Ejemplo** Los residuales para el ejemplo 3.1:

$Residual_{m0} = \{\ \}$
$Residual_{m1} = \{\ \}$
$Residual_{m2} = \{\ \{return, 2n+1\}\ \}$
$Residual_{m3} = \{\ \{return, k^2+2k\}\ \}$

□

Notar en el ejemplo anterior, que una familia es descripta como $\{return, size\}$ y no se especifican todos sus miembros. Esto se debe a que el análisis de escape propuesto utiliza los parámetros y el *return* para vincular una familia en el método llamado con las variables del método llamador. Luego, para inferir la evolución del residual alcanza con describir una familia en función de los parámetros, el *return* y una expresión de tamaño. Es importante destacar que esto restinge las posibilidades del análisis respecto a la inferencia de punteros entre objetos. Es decir, al considerar que los objetos de una familia escapan en conjunto el análisis es poco preciso ya que no puede determinar relaciones más específicas entre los objetos residuales y los objetos del método llamador. Existen diferentes análisis que permiten un cálculo más detallado de las relaciones entre objetos, por ejemplo [Sal, BFGL07]. La evaluación de otras técnicas de análisis de escape constituyen una línea de trabajo futuro.

Por último, para un punto de invocación $lis = m.l.m'$ y la especificación del residual definida debemos calcular las expresiones $C^m_{lis}$ y $R^m_{lis}$. Para esto,

determinamos que familias del residual de $m'$ son capturadas por $m$ y cuales escapan a su vez al contexto de $m$, lo que es resuelto por el análisis de escape presentado anteriormente, y obtenemos la suma de las expresiones asociadas a cada familia.

$$C^m_{lis} = \sum \{Size_F(P_{m'}) \mid F \in Residual_{m'} \wedge \neg escapa(m, F)\}$$

$$R^m_{lis} = \sum \{Size_F(P_{m'}) \mid F \in Residual_{m'} \wedge escapa(m, F)\}$$

donde F es una familia que pertenece al residual de $m'$, $Size_F(P_{m'})$ es la expresión de cardinalidad de F y $escapa(m, F)$ es un predicado, resuelto por el análisis de escape, que determina si la familia es capturada o no por el método $m$.

Cuando analizamos un método $m$ además de calcular información de consumo para $m$ debemos inferir el resumen de $m$ que permite al cálculo composicional analizar aquellos métodos que invocan a $m$. Recordemos que, dado un método $m$, el análisis de escape utilizado determina una partición de las variables de $m$ en familias. A su vez, determinamos en un punto de invocación $lis = m.l.m'$, cuales de las familias de $m'$ son capturadas por $m$ y cuales no. Esto es resuelto por el algoritmo de escape conectando las familias de $m$ con las familias de $m'$ (ver figura 5.1). Esta conexión determina que los objetos de ambas familias están relacionados y por consiguiente son ubicados en la misma región. Partiendo de esta información, para cada par de familias conectadas $f_m \rightarrow f_{m'}$ podemos aumentar el tamaño de $f_m$ en función del tamaño de $f_{m'}$.

Para incrementar el tamaño de una familia de $m$ en función de una familia residual de $m'$ debemos considerar dos cosas, el punto de invocación $lis$ que determina la conexión entre familias puede estar en un ciclo y la expresión que cuantifica la familia residual de $m'$ está en términos de $P_{m'}$. Considerando el invariante de $lis$ que determina el espacio de iteración de $lis$ y liga las variables locales de $m$ con los parámetros de $m'$ y utilizando la operación de suma sobre un invariante (ver sección 4.1.2) podemos calcular cuantos objetos son acumulados en la familia de $m$ como consecuencia de invocar a $m'$. Notar que esta operación obtiene una expresión en términos de $P_m$ como es requerido.

El algoritmo 6 actualiza las expresiones de tamaño de las familias de un método teniendo en cuenta las llamadas que éste realiza. La función $FamiliasConectadas(lis)$ devuelve un conjuto de elementos $f_m \rightarrow f_{m'}$ que indica que $f_m$ está conectada con $f_{m'}$ por la llamada que $lis$ especifica.

Finalmente, podemos describir el residual de un método $m$ utilizando el resumen presentado anteriormente y actualizando las expresiones de tamaño de las familias

**Algoritmo 6** Inferencia del residual de $m$

$total \leftarrow 0$
**for** $lis = m.l.m' \in LIS_m$ **do**
  **for** $f_m \rightarrow f_{m'} \in FamiliasConectadas(lis)$ **do**
    $Size_{f_m} \leftarrow Size_{f_m} + \mathcal{O}(\mathcal{I}^m_{lis}, Size_{f_{m'}}, P_m)$
  **end for**
**end for**

del residual mediante el algoritmo 6.

# Capítulo 6

# Implementación

En este capítulo vamos a discutir los aspectos técnicos más relevantes sobre la herramienta que implementamos para la evaluación del enfoque composicional para el análisis de consumo. Como describimos en secciones anteriores (ver figura 1.1) la solución está compuesta por cuatro grandes componentes:

- Generador de invariantes locales.
- Analizador de información de escape.
- Calculadora de expresiones paramétricas.
- Analizador de memoria dinámica.

## 6.1. Generador de invariantes

Este componente es el encargado de obtener los invariantes para los puntos relevantes del programa requeridos por las técnicas de inferencia de consumo. Los invariantes pueden ser especificados utilizando la *api* Java que provee la herramienta o mediante un archivo con formato xml (el formato es descripto en el apéndice A.1.1).

A efectos de incrementar la automatización del análisis, se integró la herramienta implementada en [Gar05] que infiere los invariantes requeridos de menera automática. Para realizar esta integración se implementaron un conjunto de clases Java que traducen la salida de la herramienta desarrollada en [Gar05] al formato esperado por nuestro prototipo.

Si bien el generador de invariantes es un componente que implementa una función específica dentro de la solución propuesta, está desacoplado de los demás componentes. Es posible correr una aplicación que utiliza este componente y genera el archivo xml requerido por el análisis de consumo, permitiendo esto, correr el análisis de consumo de forma independiente a la generación de invariantes. El apéndice A.1 describe la utilización de esta herramienta.

## 6.2. Analizador de información de escape

Este componente es responsable de determinar los tiempos de vida de los objetos que se cuantifican y de establecer la relación entre los objetos que exceden el alcance de un método.

El prototipo está conectado con la herramienta desarollada en [Sal08] que implementa el análisis de escape presentado en la sección 5.3. Esta herramienta es extendida con relaciones de tamaño (expresiones paramétricas) que permiten cuantificar los conjuntos de objetos determinados por el análisis de escape.

Si bien la información requerida por el *Analizador de consumo* podría ser determinada por distintos algoritmos, esta primer implementación no permite reemplazar el componente responsable del análisis de escape de manera trivial. Para lograr esto, hace falta la implementación de una capa que permita abstraer el resultado del análisis, lo que constituye una posible línea de trabajo a futuro (ver sección 8.2).

## 6.3. Calculadora de expresiones paramétricas. Jbarvinok.

La calculadora de expresiones paramétricas es la parte de la herramienta que requirió más trabajo. Es responsable de proveer un entorno para la operación de expresiones paramétricas. Como mencionamos anteriormente, estas operaciones son provistas por la biblioteca *libarvinok*, la cual fue integrada a nuestro prototipo.

La integración de *libbarvinok* a un componente Java no es un paso trivial ya que la biblioteca fue desarrollada en C/C++. La integración fue resuelta mediante la utilización de una herramienta llamada SWIG [Bea].

SWIG es una herramienta de desarrollo que permite conectar programas escritos en C/C++ con una variedad de lenguajes de alto nivel, en particular Java. La figura 6.1 muestra la arquitectura propuesta para la integración de *libbarvinok* a nuestro prototipo. El resultado de la integración es una biblioteca Java, independiente de la herramienta desarrollada para el cálculo composicional, denominada *Jbarvinok* que permite manipular expresiones paramétricas.

### 6.3.1. Diseño y funcionalidades

Este componente fue diseñado como una biblioteca y permite el cálculo de operaciones sobre el modelo de politopos paramétricos. Provee las funcionalidades básicas para la resolución de los problemas de cuantificación del uso de memoria (ver sección 2.1). Las tres operaciones principales son, la enumeración de un dominio, la maximización de un polinomio sobre un dominio determinado y la suma de evaluar un polinomio en todos los puntos enteros que satisfacen un conjunto de restricciones.

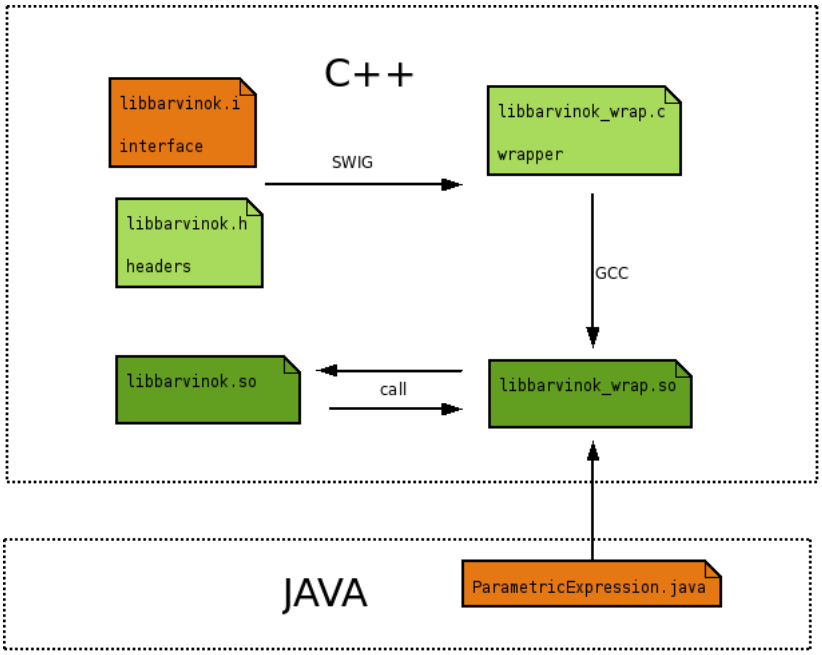


Figura 6.1: Arquitectura propuesta para la integración de libbarvinok a Java.

**Funcionalidades provistas por Jbarvinok** :

- Implementación de dominios por restricciones lineales.
- Enumeración de un dominio.
- Integración de objetos *evalue* (objetos *libbarvinok*).
- Implementación de expresiones paramétricas.
- Maximización de una expresión paramétrica sobre un dominio.
- Suma de una expresión paramétrica sobre un dominio.

Es importante destacar que el desarrollo está 100 % desacoplado del resto de la solución y por lo tanto puede ser utilizada como biblioteca en otras soluciones Java. Más aún, puede ser reemplazada por otras implementaciones sin mayor esfuerzo.

La figura 6.2 presenta las clases principales que componen la biblioteca desarrollada. Notar que para la utilización de otra biblioteca de manipulación de expresiones simbólicas sólo es necesario implementar las interfaces *Domain* y *ParametricExpression*.

## 6.4. Analizador de memoria dinámica

Este componente implementa el algoritmo composicional para la inferencia de requerimientos de memoria. El componente fue construído sobre un framework para

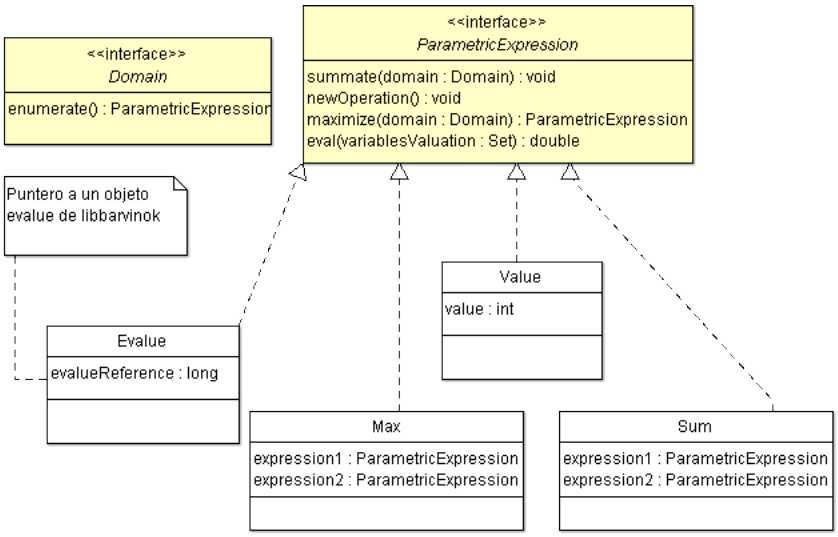


Figura 6.2: Clases principales de la biblioteca Jbarvinok.

optimización de programas Java llamado Soot [VRHS+99]. Usamos este framework para generar árboles de llamadas e implementar varios análisis de flujo de datos.

La inferencia de requerimientos de consumo es implementada mediante el algoritmo presentado en la sección 4.2. Para esto se utiliza las funcionalidades de Soot para generar el árbol de llamadas y procesar cada uno de los métodos del programa. Para determinar el consumo de un punto de creación o invocación, se utiliza el invariante determinado por el *Generador de invariantes* para ese punto y se obtiene una expresión de consumo utilizando la *Calculadora de expresiones paramétricas*.

Finalmente, se genera un salida que presenta la especificación obtenida para cada uno de los métodos analizados.

# Capítulo 7

# Experimentos y Resultados

Para evaluar tanto la técnica como la herramienta desarrollada en esta tesis se realizaron experimentos sobre el ejemplo presentado en [BFGY08] y sobre programas del *benchmark* JOlden [CM01]. Para contrastrar los resultados obtenidos por el análisis compisicional se utilizó la la herramienta que implementa la técnica presentada en [BFGY08] y la implementación de un modelo de regiones [Tab09].

## 7.1. Primer ejemplo

El primer ejemplo que vamos a considerar es el programa 7.1 presentado en [BFGY08]. Este programa es interesante porque permite analizar dos temas importantes de la técnica de análisis de consumo presentada. El primero de estos es la influencia del algoritmo de análisis de escape utilizado sobre la precisión en la inferencia de consumo. El segundo tema es la manipulación simbólica de polinomios. En particular vamos a considerar la elección de un polinomio que maximice el consumo de memoria entre un conjunto de polinomios.

La tabla 7.1 presenta los valores obtenidos por el algoritmo 4.3.2 para los distintos métodos que constituyen el programa.

| Método | TmpLocal | TmpCall | MaxCall | ResLocal | ResCall | liveObjects |
|---|---|---|---|---|---|---|
| m4 | 0 | 0 | 0 | 2 | 0 | 2 |
| m3 | 1 | $n$ | 0 | 1 | $n$ | $2n+1$ |
| m2 | 0 | $2k+2$ | 0 | 0 | 0 | $2k+2$ |
| m1 | 1 | $mc^2+3mc$ | 0 | 0 | 0 | $mc^2+3mc+1$ |
| m0 | 0 | 0 | máx $\{mc^2+3mc+1, 6mc+2\}$ | 0 | 0 | máx $\{mc^2+3mc+1, 6mc+2\}$ |

Cuadro 7.1: Resumen de consumo por método.

```
void m0(H h) {
    h.m1();
    h.m2(3 * h.mc);
}

public class N {

    B value;
    N next;

    B m4(int v) {
        N c = new N();
        c.value = new B(v);
        c.next = this.next;
        this.next = c;
        return c.value;
    }
}

public class H {

    int mc;

    void m1() {
        B[][] dummyArr = new B[
            this.mc][];
        for (int i = 1;i <= this.
            mc; i++) {
            dummyArr[i-1]= m3(i);
        }
    }

    void m2(int k2) {
        B[] m3Arr = m3(k2);
    }

    B[] m3(int n) {
        B[] arrB = new B[n];
        N l = new N();
        for (int j=1;j <= n;j++) {
            arrB[j-1] = l.m4(j);
        }
        return arrB;
    }
}
```

### 7.1.1. Análisis de escape

Los resultados obtenidos para el ejemplo anterior son correctos y reflejan de manera exacta el residual y temporal de los distintos métodos utilizando un esquema por regiones inferidas por el método de análisis de escape presentado en la sección 5.3. Sin embargo, los objetos de tipo $N$ creados durante la ejecución de $m3$ son considerados residuales. Esto se debe a que el análisis de escape utilizado no es muy preciso.

La técnica presentada en [BFGY08] obtiene una estimación menor de requerimiento de consumo ya que está basada en un análisis de escape más preciso y para el caso particular de $m3$ determina el tiempo de vida exacto de los objetos creados durante su ejecución.

La tabla 7.2 compara la estimación de $liveObjects_{m0}$ obtenida por el algoritmo composicional ($liveObjects = \max\{mc^2 + 3mc + 1,\ 6mc + 2\}$) con la estimación ($memRq_{m0} = \max\{6mc + 2, \frac{1}{2}mc^2 + \frac{5}{2}mc + 2\}$) obtenida por la técnica [BFGY08]. La columna $\frac{1}{2}mc^2$ indica el valor que toma esta expresión para los distintos valores de mc. Notar que dicha expresión es el término que domina la diferencia entre ambas estimaciones y proviene del residual de $m3$.

| mc | $memRq_{m0}$ | $liveObjects_{m0}$ | $liveObjects_{m0} - memRq_{m0}$ | $\frac{1}{2}mc^2$ |
|---|---|---|---|---|
| 10 | 77 | 131 | 54 | 50 |
| 50 | 1377 | 2651 | 1274 | 1250 |
| 150 | 11627 | 22951 | 11324 | 11250 |
| 200 | 20502 | 40601 | 20099 | 20000 |
| 500 | 126252 | 251501 | 125249 | 125000 |

Cuadro 7.2: Valores estimados para distintas valuaciones de mc.

### 7.1.2. Máximo entre polinomios

El resumen calculado para un método $m$ es utilizado para inferir el resumen de todos los métodos que invocan a $m$. En nuestro prototipo esto requiere la simplificación de las expresiones del tipo $max(e_1, \ldots, e_n)$. Determinar el máximo entre dos o más polinomios es un problema complejo. Una posibilidad es partir el dominio de las variables de los polinomios y determinar un máximo en cada subdominio. Esto no es una operación trivial y constituye una potencial línea de trabajo a futuro.

Para simplificar estas expresiones, esta primer implementación resuelve dicha operación definiendo un nuevo polinomio que determina una cota superior de los candidatos a ser el máximo. Para los polinomios de una variable esta cota es determinada por el mayor término de cada uno de los grados. Para el ejemplo anterior la cota obtenida es $mc^2 + 6mc + 2$. La figura 7.1 presenta, para diferentes valores de $mc$, el error en la estimación de $liveObjects_{m0}$ producido por dicha cota.

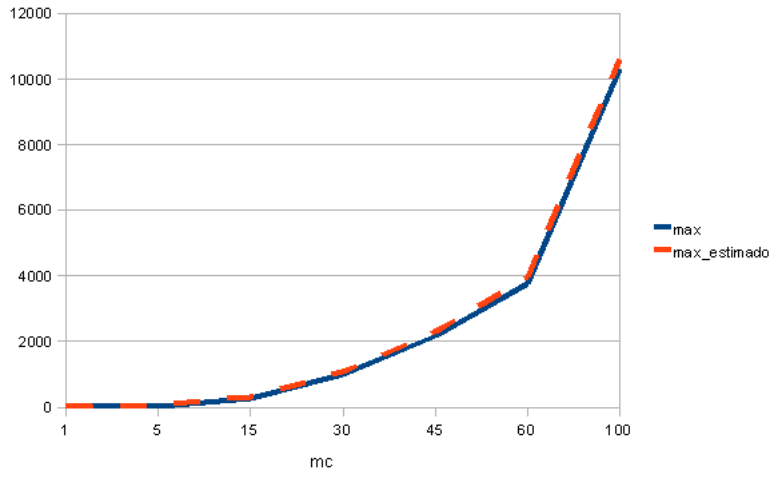


Figura 7.1: Imprecisión en la determinación del polinomio máximo

## 7.2. Recursión

Una limitación de la técnica presentada en [BFGY08] es la imposibilidad de analizar métodos recursivos. El análisis composicional permite incorporar ciertos

patrones de recursión siempre y cuando el programador especifique el resumen del método recursivo. Para evaluar un patrón de recursión simple utilizamos el programa BiSort del *benchmark* JOlden.

El código 7.1 presenta el método recursivo de la aplicación BiSort. Por cada ejecución el método crea un objeto del tipo *Value* y realiza dos llamadas recursivsa para construir el árbol derecho e izquierdo. Este método crea $2^{floor(log_2(s))}$ objetos donde $s$ es el parámetro de la aplicación.

```
static Value createTree(int size, int seed) {
    if (size > 1) {
      seed = random(seed);
      int next_val = seed mod RANGE;

      Value retval = new Value(next_val);
      retval.left = createTree(size/2, seed);
      retval.right = createTree(size/2, skiprand(seed, size+1));
      return retval;
    } else {
      return null;
    }
  }
```

Listing 7.1: BiSort

Para inferir el consumo de los métodos que invocan al método recursivo *creaTree* podemos enriquecer el análisis con la información de consumo de este método. Esto se logra especificando el valor del residual que *createTree* genera en el llamador. Siendo que la ténica presentada soporta expresiones polinomiales tomamos como una aproximación del residual $2^{floor(log_2(s))}$ a la expresión lineal $s$. La tabla 7.3 presenta los valores obtenidos por el algoritmo composicional a partir de la especificación del residual del método *createTree*.

| Método | TmpLocal | TmpCall | MaxCall | ResLocal | ResCall | liveObjects |
|---|---|---|---|---|---|---|
| createTree | 0 | 0 | 0 | $s$ | 0 | $s$ |
| parseCmdLine | 2 | 0 | 0 | 0 | 0 | 2 |
| mainParameters | 5 | 0 | 1 | 0 | $s$ | $s+6$ |

Cuadro 7.3: Resumen de consumo por método para BiSort.

## 7.3. JOlden

Además de la aplicación BiSort evaluamos los programas MST y EM3D del *benchmark* JOlden. Para estos programas se calcularon los resumenes por método utilizando el algoritmo composicional. La estimación de consumo del método principal de

ambos programas es comparada con los valores obtenidos en ejecuciones de las aplicaciones utilizando el modelo de regiones implementado en [Tab09].

### 7.3.1. MST

Esta aplicación construye un grafo $G$ cuyo tamaño es en función del parámetro $numVert$ y ejecuta el algoritmo $computeMST$ que obtiene el árbol de expansión mínima de $G$. La tabla 7.4 presenta las estimaciones de consumo para los métodos relevantes de la aplicación obtenidas por el algoritmo composicional.

| Método | TmpLocal | TmpCall | MaxCall | ResLocal | ResCall | liveObjects |
|---|---|---|---|---|---|---|
| doAllBlueRule | 0 | 0 | 0 | 0 | 1 | 1 |
| computeMST | 6 | 0 | 0 | 0 | $numvert$ | $numvert$ |
| addEdges | 0 | 0 | 0 | $2 \cdot numvert^2$ | 0 | $2 \cdot numvert^2$ |
| parseCmdLine | 2 | 0 | 0 | 0 | 0 | 2 |
| mainParameters | 6 | $2 \cdot numvert^2 + 4 \cdot numvert + 1$ | 0 | 0 | 0 | $2 \cdot numvert^2 + 4 \cdot numvert + 7$ |

Cuadro 7.4: Resumen de consumo por método para MST.

La tabla 7.5 presenta el valor estimado de consumo y el consumo real del método $mainParameters$ en función del parámetro $numvert$.

| numvert | $liveObjects$ estimado | $liveObjects$ | error |
|---|---|---|---|
| 50 | $2 \cdot numvert^2 + 4 \cdot numvert + 7 = 5207$ | 5150 | $1,11\,\%$ |
| 75 | $2 \cdot numvert^2 + 4 \cdot numvert + 7 = 11557$ | 11475 | $0,71\,\%$ |
| 100 | $2 \cdot numvert^2 + 4 \cdot numvert + 7 = 20407$ | 20300 | $0,53\,\%$ |

Cuadro 7.5: Error en la estimación de consumo para MST.

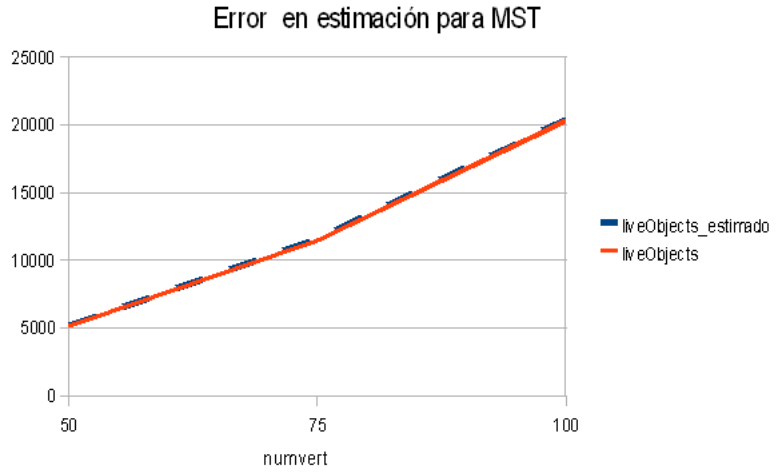


Figura 7.2: Error en la aproximación de consumo para MST

### 7.3.2. EM3D

Al igual que la aplicación MST, el programa EM3D construye un grafo $G$ en función del parámetro *numNodes*. Luego, ejecuta *numIter* veces un algoritmo sobre $G$. La tabla 7.6 presenta las estimaciones de consumo para los métodos relevantes de la aplicación EM3D obtenidas por el algoritmo composicional.

| Método | TmpLocal | TmpCall | MaxCall | ResLocal | ResCall | liveObjects |
|---|---|---|---|---|---|---|
| create | 0 | 0 | 0 | $8 \cdot numNodes + 9$ | 0 | $8 \cdot numNodes + 9$ |
| parseCmdLine | 6 | 0 | 0 | 0 | 0 | 6 |
| compute | 0 | 0 | 0 | 2 | 0 | 2 |
| mainParameters | 4 | $8 \cdot numNodes + 2 \cdot numIter + 9$ | 0 | 0 | 0 | $8 \cdot numNodes + 2 \cdot numIter + 13$ |

Cuadro 7.6: Resumen de consumo por método para EM3D.

La tabla 7.7 presenta el valor estimado de consumo y el consumo real del método *mainParameters* en función del parámetro *numNodes* para un valor fijo de *numIter* igual a 1.

| numvert | *liveObjects* estimado | *liveObjects* | error |
|---|---|---|---|
| 20 | $8 \cdot numNodes + 15 = 175$ | 159 | $10,06\,\%$ |
| 40 | $8 \cdot numNodes + 15 = 335$ | 319 | $5,02\,\%$ |
| 60 | $8 \cdot numNodes + 15 = 495$ | 479 | $3,34\,\%$ |
| 80 | $8 \cdot numNodes + 15 = 655$ | 639 | $2,5\,\%$ |
| 100 | $8 \cdot numNodes + 15 = 815$ | 799 | $2\,\%$ |
| 120 | $8 \cdot numNodes + 15 = 975$ | 959 | $1,67\,\%$ |

Cuadro 7.7: Error en la estimación de consumo para EM3D.

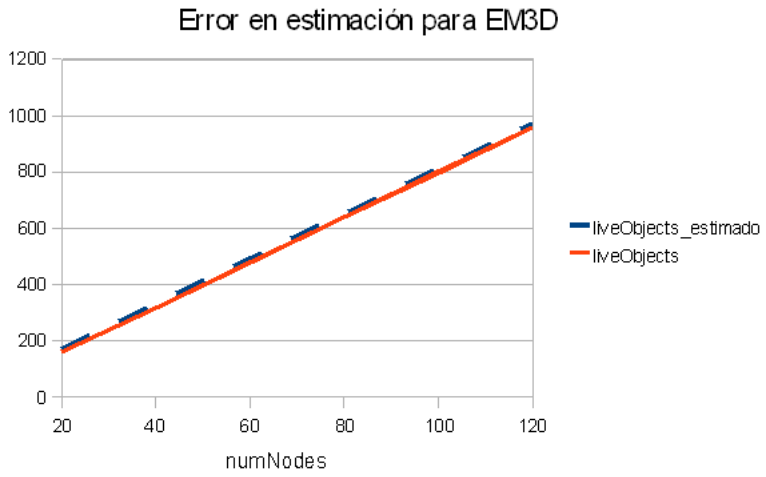


Figura 7.3: Error en la aproximación de consumo para EM3D

# Capítulo 8

# Conclusiones

Hemos presentado un algoritmo composicional que infiere resumenes por método que especifican el efecto, desde el punto de vista del consumo de memoria, de la ejecución de un método. El resumen describe de forma cuantitativa la cantidad de objetos temporales requeridos por la ejecución de un método. A su vez, describe conjuntos de objetos residuales asociados a una expresión en términos de los parámetros del método que aproxima el tamaño de los mismos.

Las especificaciones obtenidas para cada método pueden ser utilizadas como base para otros análisis para aproximar el efecto de las llamadas a métodos. A su vez pueden ser utilizadas para implementar políticas de administración de memoria eficientes (por ejemplo [Tab09]).

En comparación con las técnicas desarrolladas en [BGY06, BFGY08] podemos decir que el análisis composicional presenta algunas ventajas. Una de ellas es el incremento de la escalabilidad, ya que este nuevo enfoque requiere analizar estados locales en vez de globales. A su vez, incrementa la usabilidad permitiendo analizar cierta clase de métodos que antes no se podían analizar tales como métodos recursivos y métodos para los cuales es complejo obtener los invariantes requeridos por las técnicas de inferencia de consumo. Esto se debe a que el comportamiento de un método puede ser especificado por el programador sin necesidad de recurrir al uso de nuestra herramienta.

Una contribución importante de este trabajo es la integración de técnicas de análisis de consumo con técnicas de análisis de escape, en particular utilizamos el análisis presentado en [SYG05]. Esto permite que el análisis de consumo pueda ser realizado con menor intervención del programador.

Finalmente, es importante destacar que el enfoque presentado fue evaluado en una herramienta implementada a lo largo de esta tesis. La implementación de esta herramienta no es algo trivial ya que combina herramientas de análisis estático y dinámico y manipulación simbólica de poliedros y polinomios.

## 8.1. Limitaciones

Si bien este nuevo enfoque permite aumentar el conjunto de programas analizables todavía hay casos que no tratamos, por ejemplo algunos patrones de recursión. Por otro lado, los invariantes utilizados deben seguir siendo lineales y numéricos.

Existen diferentes fuentes de imprecisión intrínsecas al enfoque presentado. La utilización de invariantes locales, si bien son más sencillos y manipulables que los invariantes globales, son una potencial fuente de imprecisión ya que proporcionan menor información de contexto que los invariantes globales. Un invariante global es necesariamiente más preciso.

En la sección 5.2.4 mostramos que para maximizar una expresión del tipo $max(expresion1, expresion2)$ en función de un invariante, si no podemos determinar el máximo entonces tomamos una cota superior de esta expresión. Esta aproximación puede producir cotas demasiado pesimistas.

Otro elemento en donde podemos realizar mejoras es en el cálculo del tiempo de vida de los objetos. El análisis de escape utilizado fue diseñado para ser sobre todo eficiente relegando cierto grado de precisión. En general, como todo análisis estático, no determina tiempos de vida exactos sino aproximados. En particular, este análisis considera a los punteros de forma bidireccional. Como consecuencia, varios objetos que son temporales son considerados parte del residual generando mayor consumo.

## 8.2. Trabajos a futuro

Hay varios aspectos de nuestra técnica que quisieramos mejorar. Un aspecto importante es la imprecisión generada por la cota utilizada para aproximar el máximo entre dos expresiones cuando no sabemos determinar cuál de éstas es la mayor. Como fue mencionado en la sección 5.2.4 una posible solución es distribuir la operación $maxInv$ y continuar el cálculo. Esto requiere la implementación de expresiones paramétricas más complejas y seleccionar la estrategia más conveniente (aproximar el máximo o distribuir la operación $maxInv$). A su vez, la cota utilzada para aproximar el máximo puede ser refinada.

Otro aspecto sobre el cuál quisieramos dedicar esfuerzo es en no restringir el cálculo composicional al uso del análisis de escape presentado. De hecho, estamos trabajando en una formalización de la técnica presentada que permite abstraer la técnica de análisis de escape utilizada.

Soportar métodos recursivos es una asignatura pendiente. Si bien el análisis composicional permite incluír cierto tipo de métodos recursivos, depende de la especificación del método por parte del programador. Para soportar patrones más generales de recursión quisieramos utilizar el enfoque presentado en [AGGZ09] basado en ecuaciones recurrentes.

La precisión de nuestra técnica depende, en gran parte, de proveer invariantes suficientemente fuertes. Para tratar esto presentamos diferentes alternativas, generar los invariantes automáticamente, especificarlos de forma manual o una combinación de ambas. Soportar la utilización de anotaciones JML [LLP+00] es una característica interesante que quisieramos incluir, ya que permite al programador anotar el código y a su vez permite validar su correctitud.

# Bibliografía


[AAG+07] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of java bytecode. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2007.

[AGGZ09] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Live heap space analysis for languages with garbage collection. In *ISMM*, pages 129–138, 2009.

[AM05] David Aspinall and Kenneth MacKenzie. Mobile resource guarantees and policies. In Gilles Barthe, Benjamin Grégoire, Marieke Huisman, and Jean-Louis Lanet, editors, *CASSIS*, volume 3956 of *Lecture Notes in Computer Science*, pages 16–36. Springer, 2005.

[BCG04] D. Bacon, P. Cheng, and D. Grove. Garbage collection for embedded systems. *In EMSOFT'04*, 2004.

[Bea] David Beazley. An interface compiler to connect c/c++ programs with scripting languages.
Available at http://www.swig.org/.

[BFGL07] Mike Barnett, Manuel Fändrich, Diego Garbervetsky, and Francesco Logozzo. Annotations for (more) precise points-to analysis. In *IWACO 2007: ECOOP International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, Berlin, Germany, jul 2007.

[BFGY08] Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements, jun 2008.

[BG00] Greg Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[BGY06] Víctor A. Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

[BHMS04] Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In *LPAR*, pages 347–362, 2004.

[Bla99] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *OOPSLA 99*, volume 34, pages 20–34, 1999.

[BP98] Gianfranco Bilardi and Keshav Pingali. The static single assignment form and its computation. Technical report, 1998.

[BPS05] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 86–95. IEEE Computer Society, 2005.

[Bro84] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Symposium on LISP and functional programming*, pages 256–262. ACM Press, 1984.

[CEI+07] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. *ACM Trans. Program. Lang. Syst.*, 29(5):28, 2007.

[CFGV09] Philippe Clauss, Federico Fernández, Diego Garbervetsky, and Sven Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. *IEEE Transactions on Very Large Scale Integration System*, 2009.

[CJPS05] David Cachera, Thomas P. Jensen, David Pichardie, and Gerardo Schneider. Certified memory usage analysis. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2005.

[CKQ+05] W. Chin, S. Khoo, S. Qin, C. Popeea, and H. Nguyen. Verifying safety policies with size properties and alias controls. In *ICSE 2005*, 2005.

[CL98] Ph. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19(2):Kluwer Academic, 1998.

[Cla96] P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *ICS'96*, pages 278–285, 1996.

[CM01] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java controller. In *PACT 2001*, pages 280–291, 2001.

[CNQR05] W. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *SAS 05*, 2005.

[CR04] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. *ISMM'04*, 2004.

[CR07] Sigmund Cherem and Radu Rugina. A practical escape and effect analysis for building lightweight method summaries. In *CC 2007: 16th International Conference on Compiler Construction*, Braga, Portugal, March 2007.

[Fah98] Thomas Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *J. Supercomput.*, 12(3):227–252, 1998.

[Fea96] P. Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS*, chapter Automatic Parallelization in the Polytope Model, pages 79–100. Springer-Verlag, 1996.

[Gar05] D. Garbervetsky. Using daikon to automatically estimate the number of excecuted instructions. *Internal Report. UBA, Argentina*, 2005.

[Ghe02] O. Gheorghioiu. Statically determining memory consumption of real-time java threads. MEng thesis, Massachusetts Institute of Technology, June 2002., 2002.

[GNYZ04] Diego Garbervetsky, Chaker Nakhli, Sergio Yovine, and Hichem Zorgati. Program instrumentation and run-time analysis of scoped memory in Java. *RV 04, ETAPS 2004, ENTCS, Barcelona, Spain*, April 2004.

[GS00] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 82–93, London, UK, 2000. Springer-Verlag.

[Hen98] R. Henriksson. Scheduling garbage collection in embedded systems. *PhD. Thesis, Lund Institute of Technology*, 1998.

[HIB+02] T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J-Ph. Lesot, and F. Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

[HJ03] M. Hofman and S. Jost. Static prediction of heap usage for first-order functional programs. In *POPL 03*, SIGPLAN, New Orleans, LA, January 2003.

[HP99] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ml programming. In *ICFP '99*, pages 70–81. ACM, 1999.

[JL96] R. Jones and R. Lins. *Garbage collection. Algorithms for automatic dynamic memory management.* John Wiley and Sons, 1996.

[LLP+00] G.T. Leavens, K. Rustan M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA'00*, pages 105–106, 2000.

[PFHV04] Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time Java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, Vienna, Austria, May 2004.

[RF02] T. Ritzau and P. Fritzon. Decreasing memory over-head in hard real-time garbage collection. *In EMSOFT'02, LNCS 2491*, 2002.

[Sal] Alexandru Salcianu. Pointer analysis and its applications for java programs. SM Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 2001.

[Sal08] Guillaume Salagnac. *Synthèse de gestionnaires mémoire pour applications Java temps-réel embarquées.* PhD thesis, Université Joseph-Fourier - Grenoble I, 04 2008.

[Sie00] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. *CASES'00*, 2000.

[SR01] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP 01*, volume 36, pages 12–23, 2001.

[SR05] Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, January 2005.

[SYG05] Guillaume Salagnac, Sergio Yovine, and Diego Garbervetsky. Fast escape analysis for region-based memory management. *Electronic Notes Theoretical Comput. Sci.*, 131:99–110, 2005.

[Tab09] Alejandro Taboada. *Implementación de un modelo de memoria basado en regiones en una máquina virtual a gran escala.* PhD thesis, jul 2009.

[TT97] M. Tofte and J.P. Talpin. Region-based memory management. *Information and Computation*, 1997.

[USL03] L. Unnikrishnan, S.D. Stoller, and Y.A. Liu. Optimized live heap bound analysis. In *VMCAI 03*, volume 2575 of *LNCS*, pages 70–85, January 2003.

[VB08] Sven Verdoolaege and Maurice Bruynooghe. Algorithms for weighted counting over parametric polytopes: A survey and a practical comparison. In Matthias Dehmer, Michael Drmota, and Frank Emmert-Streib, editors, *ITSL*, pages 60–66. CSREA Press, 2008.

[Ver07] Sven Verdoolaege. *barvinok*, a library for counting the number of integer points in parametrized and non-parametrized polytopes.
Available at http://freshmeat.net/projects/barvinok, April 2007.

[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.

[VSB+04] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In *Proceedings of International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 248–258, September 2004.

# Apéndice A

# Uso de las herramientas

Este apéndice presenta los comandos necesarios para utilizar la herramienta de cálculo de consumo desarrollada en esta tesis. Para ejemplificar su uso vamos a considerar el siguiente ejemplo (A.1).

```
public class EjemploGenInv {

    public static void main(String[] args) {
        int args0 = Integer.parseInt(args[0]);
        EjemploGenInv ejemploGenInv = new EjemploGenInv();
        for(int i=1;i<=args0;i++)
            ejemploGenInv.sumar(i);
    }

    public int sumar(int n) {
        Integer[] newVector = this.vector(n);
        int suma = 0;
        for(int i=0;i<n;i++) {
            suma += newVector[i].intValue();
        }
        return suma;
    }

    public Integer[] vector(int n) {
        Integer[] vector = new Integer[n];
        for(int i=0;i<n;i++)
            vector[i] = new Integer(i);
        return vector;
    }
}
```

Listing A.1: Ejemplo guía para el cálculo de consumo.

Para facilitar el uso de las distintas herramientas utilizadas en esta tesis, se configuró un entorno virtual utilizando VirtualBox. Este entorno provee las librerías compiladas, un entorno de desarrollo Java y las herramientas implementadas durante

este trabajo. En este entorno, el ejemplo A.1 se encuentra en *madeja/tests/tesis/ejemplos/*.

## A.1. Generación de invariantes

En esta sección se presenta la forma de utilización de la herramienta para la generación de invariantes. La salida generada por la herramienta es el archivo *sites.xml* que contiene la descripción de los puntos de creación de objetos e invocaciones a los distintos métodos del programda. La sintaxis del comando es:

```
./inv.sh nombreClase arg0, arg1, ....,argN
```

Donde *nombreClase* corresponde a la sintáxis Java para identificación de una clase, para el ejemplo presentado *tesis.ejemplos.EjemploGenInv* y arg0, arg1, ..., argN son los argumentos del programa a analizar.

La herramienta utiliza clases compiladas la cuales deben estar en el classpath. En general vamos a utilizar como entrada del programa clases compiladas dentro del proyecto mismo de la herramienta, sin embargo es posible utilizar otro origen. En este caso debemos modificar el sript *./inv.sh* para agregar al classpath el origen de los .class a ser analizados.

En el ejemplo presentado, el método raíz que nos interesa analizar es sumar. La herramienta de generación de invariantes requiere ejecutar varias veces el programa bajo análisis, luego agregamos un método main que realiza un ciclo y ejecuta el método sumar con distintos valores para la obtención de invariantes.

La salida generada es un archivo xml que contiene la información de los sitios de creación y los sitios de invocación a métodos. Todos los script generan las salidas en una carpeta *target/clase*, siendo *clase* la sintáxis Java para identificación de la clase en análisis. En este caso particular el resultado se encontrará en *target/tesis.ejemplos.EjemploGenInv/sites.xml*.

### A.1.1. Sobre el formato de salida

La información generada por el *Generador de invariantes* de invariantes es reflejada en un arhivo (*sites-xml*) con el siguiente formato:

```
<sites>
   <creation-site>
      <id type="declarative" method="methodDescriptor"
          bytecode-offset="bo" srccode-offset="so"/>
      <relevant-variables>v1,...vn</relevant-variables>
      <relevant-parameters>p1,..,pn</relevant-parameters>
      <type>type</type>
      <constraints>
         <constraint>c1</constraint>
             ......
         <constraint>cn</constraint>
      </constraints>
   </creation-site>
   <call-site>
      <id type="declarative" method="methodDescriptor"
          bytecode-offset="bo" srccode-offset="so"/>
      <relevant-variables>v1,...,vn</relevant-variables>
      <relevant-parameters>p1,...,pn</relevant-parameters>
      <target>methodDescriptor</target>
      <constraints>
         <constraint>c1</constraint>
             ......
         <constraint>cn</constraint>
      </constraints>
      <binding>
         <relevant-parameters>p1,....pn</relevant-parameters>
         <constraints>
            <constraint>c1</constraint>
            ......
            <constraint>cn</constraint>
         </constraints>
      </binding>
   </call-site>
</sites>
```

Listing A.2: sites.xml

Cada sitio de creación es descripto mediante un elemento xml del tipo *creation-site* compuesto por:

**id:** Declaración del método, posición del bytecode y la línea de código fuente del punto del programa.

**relavant-variables:** Variables relevantes para ese punto del programa.

**relevant-parameters:** Parámetros relevantes del método.

**type:** Clase de objeto que se crean en este punto.

**constrains:** Restricciones sobre ese punto del programa, constituyen el invariante asociado a la técnica de conteo.

Cada sitio de invocación es descripto mediante un elemento xml del tipo *call-site* compuesto por:

**id:** Declaración del método, posición del bytecode y la línea de código fuente del punto del programa.

**relavant-variables:** Variables relevantes para ese punto del programa.

**relevant-parameters:** Parámetros relevantes del método.

**target:** Método al que se invoca.

**constrains:** Restricciones sobre ese punto del programa, constituyen el invariante asociado a la técnica de conteo.

**binding:** Elemento que especifica la relación entre las variables locales y los parámetros del método target.

**relevant-parameters:** Parámtros relevantes del método target.

**constrains:** Restricciones entre las variables locales y los parámetros del método target (*binding invariant*).

### A.1.2. Modificando la información de sitios

En la salida generada del ejemplo anterior podemos ver la información obtenida para el sitio de creación *Integer[] vector = new Integer[n]*:

```
<creation-site>
  <id type="declarative" method="EjemploGenInv:Integer[]_vector(int)"
bytecode-offset="1" srccode-offset="28"/>
  <relevant-variables>__i0,n</relevant-variables>
  <relevant-parameters>n_init</relevant-parameters>
  <type>java.lang.Integer</type>
  <arguments></arguments>
  <constraints>
    <constraint>__i0 =  n</constraint>
    <constraint>__i0 =  n_init</constraint>
  </constraints>
</creation-site>
```

Listing A.3: sites.xml

La creación de arreglos de objetos es tratada como un objeto más, es decir se cuenta como un único objeto. El invariante obtenido no aporta información relevante para este conteo. Ante esta situación podemos optar por dos opciones distintas, eliminar el sitio de creación del archivo *sites.xml* o modificar el invariante. En el primer caso el análisis de consumo, al no encontrar un invariante para ese sitio de creación, lo tratará como la creación de un único objeto. El segundo caso nos permite refinar el invariante para obtener el valor esperado.

## A.2. Análisis de consumo

La herramienta utiliza dos entradas opcionales, el archivo *sites.xml* descripto en la sección anterior y el archivo *consumo.spec.xml* que será descripto más adelante. La sintáxis del comando es:

```
./consumo.sh nombreClase
```

La herramienta busca en el directorio *target/clase* los archivos *sites.xml* y *consumo.spec.xml*. El archivo *sites.xml* es utilizado para obtener el invariante de un sitio de creación y el invariante de una llamada. Estos invariantes son necesarios para las operaciones de conteo utilizadas por la técnica desarrollada. El archivo *consumo.spec.xml* permite cuantificar el temporal y residual de los métodos alcanzables por el programa bajo análisis.

La salida generada por la herramienta es el archivo *index.html* que contiene la descripción de consumo de los métodos alcanzables por el programa bajo análisis.

### A.2.1. Especificando el consumo temporal y residual

Es posible enriquecer el análisis de consumo especificando el comportamiento de un método mediante un xml de entrada. Esta especificación permite cuantificar el consumo temporal y residual de un método que forme parte del programa bajo análisis. Para generar el archivo correspondiente a la especificación debe correrse el comando:

```
./spec.sh clase
```

Esto genera en el directorio *target/clase/* el archivo *consumo.spec.xml* con la especificación de temporales y residuales por método. En el ejemplo propuesto anteriormente, el comando es ./spec.sh tesis.ejemplos.EjemploGenInv generandose la salida en *target/tesis/ejemplos.EjemploGenInv/spec.xml*. La clase corrresponde a una clase Java compilada accesible desde el classpath del script.

Supongamos que queremos indicarle al análisis que el temporal del método vector es, en vez de 0, $n$. Para esto debemos seguir los siguientes pasos:

1. Generar el archivo de especificaciones mediante el comando:

   ```
   ./spec.sh tesis.ejemplos.EjemploGenInv 20
   ```

2. Editar el archivo *consumo.spec.xml* obtenido y modificar el tag que indica el consumo del temporal para el método vector:

```
<method id="&lt;tesis.ejemplos.EjemploGenInv:
java.lang.Integer[]_vector(int)&gt;">
    <temporal>
      <size> n </size>
    </temporal>
    <residual>
      <family>
        <pfm index="this"/>
      </family>
      <family>
        <lfm new_bcode="12" line="30"/>
        <lfm new_bcode="1" line="28"/>
      </family>
    </residual>
  </method>
```

Listing A.4: sites.xml

3. Ejecutar nuevamente el análisis de consumo:

   ```
   ./consumo.sh tesis.ejemplos.EjemploGenInv
   ```

Las expresiones de consumo especificadas pueden ser o constantes o expresiones en función de los parámetros del método. Si la expresión especificada contiene variables con nombres que no están incluídas entre los parámetros formales del método, el parser del análisis generará un error.

# Apéndice B

# Jbarvinok: Un ejemplo

En la sección 2.1 se presentaron tres problemas relacionados a la optimización de programas. Estas tres operaciones son implementadas por Jbarvinok y constituyen el núcleo de las operaciones sobre expresiones paramétricas utilizadas por el análisis de consumo. A continuación se presenta el código Java necesario para resolver los ejemplos de la sección 2.1 utilizando los objetos de la librería implementada en este trabajo.

```
package ar.uba.dc.barvinok;

import java.util.Set;
import java.util.TreeSet;

import ar.uba.dc.barvinok.constraint.Constraints;
import ar.uba.dc.barvinok.util.EvalueUtils;
import ar.uba.dc.expressions.ParametricExpression;

public class SampleTest extends BarvinokTest {

  public void testEnum() {

    //Definición del dominio
    Constraints constraints = new Constraints(new String[]{"i", "j"},
        new String[]{"n"});
    constraints.newConstraint().add("i").get(1); // i >= 1
    constraints.newConstraint().add("j").get(1); // j >= 1
    constraints.newConstraint().add("n").minus("i").get(0); // i <= n
    constraints.newConstraint().add("n").minus("j").get(0); // j <= n

    //Cantidad de soluciones enteras que satisfacen
    //el conjunto de restricciones
    ParametricExpression expression = constraints.domain().enumerate();
    System.out.println(expression.asString());
    //n^2
  }
```

```
  public void testSumAndMax() {

    //Definición de  la expresión n^2
    Set vars = new TreeSet();
    vars.add("n");
    ParametricExpression src = EvalueUtils.fromString("n*n", vars);

    //Definición del dominio
    Constraints constraints = new Constraints(new String[]{"n", "i"},
        new String[]{"k"});
    constraints.newConstraint().add("i").get(1); // i >= 1
    constraints.newConstraint().add("k").minus("i").get(0); // i <= k
    constraints.newConstraint().minus("n").add("i").equal(0); // n = i

    //Obtiene la suma de n^2 sobre el dominio
    ParametricExpression sum = src.summate(constraints.domain());
    System.out.println(sum.asString());
    //1/3 * k^2 + 1/2 * k^2 + 1/6 * k

    //Maximiza la expresion n^2 sobre el dominio
    ParametricExpression max = src.maximize(constraints.domain());
    System.out.println(max.asString());
    //k^2
  }
}
```

Listing B.1: Utilización de Jbarvinok.

# Índice de figuras

# Índice de cuadros