Tesis de Licenciatura en Ciencias de la Computación:

Un análisis composicional para la inferencia de resúmenes de consumo de memoria.

Matías Grunberg

Libreta universitaria: 486/06

grunberg@gmail.com

Gastón Krasny

Libreta universitaria: 16/06

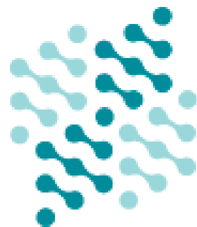
gukras@gmail.com

Director:

Dr. Diego Garbervetsky

Codirector:

Lic. Martín Rouaux



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Resumen

El estudio cuantitativo de requerimientos de memoria, es decir, predecir la cantidad de memoria necesaria para ejecutar un programa, es al día de hoy un problema desafiante. Existen diferentes técnicas para tratar dicho problema, las cuales encaran problemáticas tales como: escalabilidad, precisión, etc.

Una alternativa posible para abordar el problema de escalabilidad es definir un algoritmo composicional para la inferencia de cotas paramétricas del consumo de memoria. Por ejemplo, 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 realiza) y de la especificación de consumo de los métodos llamados por m.

Una forma de aproximar dicha cota es observar los efectos que la ejecución de m tiene sobre el *heap*, en otras palabras, analizar a los objetos creados durante su ejecución. Para esto podemos distinguir dos tipos de objetos: los creados por m y los creados por los métodos invocados por m. Los objetos pueden ser catalogados como *temporales* o *residuales* en función de su tiempo de vida. Esto permite distinguir la memoria que puede ser liberada al finalizar la ejecución de m de la que no puede serlo.

El enfoque mencionado requiere además disponer de reglas de composición que permitan relacionar la especificación de consumo de un método invocado con el punto de invocación. Es decir, estas reglas deben encargarse de resolver problemas que surgen a la hora de considerar el contexto de invocación, entre ellas aliasing y la cantidad de veces que se realiza una invocación (en presencia de un loop, etc).

Esta tesis presenta un algoritmo de análisis composicional el cual utiliza el enfoque mencionado implementando las reglas necesarias. El algoritmo es capaz de soportar diferentes instanciaciones de análisis que aproximan el tiempo de vida de los objetos, ayudando al estudio cuantitativo de requerimientos de memoria en programas Java. Además, aborda la problemática de analizar programas con llamados polimórficos, característica esencial del paradigma de programación orientado a objetos. Por último, presenta el diseño e implementación de un prototipo que fue utilizado para evaluar este algoritmo, el cual pretende mejorar la precisión de trabajos anteriores y es evaluado utilizando diferentes programas, uno de ellos de tamaño medio que involucra 36 clases.

Agradecimientos

A Diego y Martín, nuestros directores, por todo el apoyo y dedicación que nos brindaron a lo largo de todo este trabajo. A Diego, con quien compartimos muchas horas y materias durante toda la carrera, nos motivó a realizar este trabajo y nos brindó un trato de amigos desde el primer día. A Martín, quien con su experiencia reciente en este tipo de trabajos pudo guiarnos a resolver cada problema que fuimos encontrando, estos últimos meses con las dificultades que trae la distancia.

A toda la gente que compartió la carrera con nosotros durante estos 6 años, desde aquel que nos ayudó a recordar una simple regla ortográfica para realizar este trabajo, hasta aquellos con los que cursamos tantas materias en la facultad. Principalmente a Tapi y Pablo con los que compartimos tantas horas de estudio y trabajos prácticos.

A toda la gente del DC que conforma esta carrera, a todos los profesores y principalmente ayudantes que nunca tuvieron problema en quedarse hasta la hora que sea necesario para responder cualquier tipo de consultas. Estamos orgullosos de pertenecer a esta institución.

De Gastón A mis viejos Arturo y Lili, a mis hermanos Romi, Nico y Martu, a mis primas Lore y Meli, y al resto de la familia, por haber apoyado desde el primer día, por esperar siempre con la comida caliente y servida y por tolerar mi constante intolerancia, impaciencia y gritos. “MAMA, ¡LA PUERTA¡‘.

A los pibes. Es imposible hacer una carrera sin un grupo de amigos para salir, disfrutar, divertirse y olvidarse de la rutina diaria. La cena de los miércoles es algo que nunca tenemos que perder, es el segundo fin de semana que le faltaba a la semana.

A vos, blanco y negro querido, que me llevaste a recorrer el país y me demostraste que el estudio y la pasión pueden convivir. Gracias por esta alegría de primerA. Que el sueño nunca termine.

De Matías A mis papas Pablo y Graciela, mis hermanos Nico y Jeni, por haber apoyado desde el primer día, por soportar mi humor en los días complicados donde las cosas no parecían salir muy bien, por entenderme, por preocuparse y por alentarme siempre para llegar a este momento.

A mis amigos que me acompañaron siempre, en especial durante el armado de este trabajo, sabiendo entender cada vez que les dije “no puedo salir, necesito estudiar“, frase que con el correr de los meses se hizo más celebre. Gracias por estar conmigo en todo momento.

Por último, quiero agradecerle a mi novia Dalia por haberme aguantado, ayudado y apoyado durante este último año. Gracias por entender el esfuerzo que necesitó este trabajo y bancarte el tiempo que tuve que dedicarle. Gracias por acompañarme en este momento tan importante de mi vida.

Índice general

Índice general 4

1. Introducción 8
1.1. Motivación . . . 8
1.2. Acerca de este trabajo . . . 9
1.3. Visión General . . . 13
1.4. Resumen de contribuciones . . . 15
1.5. Estructura . . . 16

2. Conceptos preliminares 17
2.1. Notación para programas . . . 17
2.2. Organización de memoria predecible . . . 18
2.2.1. Tiempo de vida de un objeto . . . 19
2.2.2. Análisis de escape . . . 19
2.3. Resúmenes de consumo de memoria . . . 20

3. Ejemplos motivacionales 25
3.1. Analizando llamadas . . . 26
3.2. Conclusiones . . . 33

4. Análisis composicional del consumo de memoria 34
4.1. Infiriendo resúmenes . . . 34
4.2. Herramientas auxiliares . . . 41
4.2.1. EA . . . 41
4.2.2. CT . . . 41
4.2.3. SA . . . 42
4.3. Considerando invarianza de ciclo . . . 43
4.4. Conclusiones . . . 45

5. Análisis de escape 46
5.1. Definiciones . . . 48
5.2. Análisis del ejemplo . . . 50
5.3. Algoritmo Intra-procedural . . . 53

5.4. Algoritmo Inter-procedural . . . 54
5.4.1. Construcción del mapeo de nodos . . . 55
5.4.2. Combinando los Points-To graph . . . 56
5.5. Extendiendo hacia k-sensitivo . . . 56
5.6. Limitaciones . . . 58
5.7. Conclusiones . . . 58

6. Calculando efectivamente requerimientos de memoria 59
6.1. Expresiones Paramétricas . . . 59
6.2. Particiones de memoria . . . 60
6.3. Implementación de los componentes . . . 61
6.3.1. Implementación del CT . . . 61
6.3.2. Implementación del SA . . . 63
6.3.3. Implementación del EA . . . 68
6.4. Polimorfismo . . . 71
6.4.1. Coloreando nodos . . . 73
6.4.2. Agrupando nodos . . . 77
6.5. Limitaciones . . . 79
6.6. Conclusiones . . . 79

7. Implementación 81
7.1. Proveedor de invariantes . . . 81
7.2. Analizador de información de escape . . . 82
7.3. Calculadora de expresiones paramétricas . . . 83
7.4. Analizador de memoria dinámica . . . 84

8. Experimentos y resultados 86
8.1. Ejemplos motivacionales . . . 88
8.2. JOlden . . . 89
8.2.1. MST . . . 89
8.2.2. EM3D . . . 90
8.3. JLayer . . . 92

9. Conclusiones 94
9.1. Limitaciones . . . 95
9.2. Trabajos relacionados . . . 95
9.3. Trabajos a futuro . . . 98

Bibliografía 100

A. Uso de las herramientas **104**

A.1. Invariantes 104

A.2. Análisis de escape 105

A.3. Análisis de consumo 106

Índice de figuras **108**

Índice de cuadros **109**

Capítulo 1

Introducción

1.1. Motivación

Existe un interés creciente en entender y analizar el uso de memoria por parte de los sistemas que hacen uso intensivo de la misma o bien cuentan con recursos limitados. Por ejemplo, Cloud Computing, que permite el uso de un gran número de recursos computacionales compartidos, y los sistemas móviles o embebidos podrían resultar beneficiados al contar con un modelo predecible de consumo de memoria por parte de las aplicaciones que ejecutan. Esta información podría ser utilizada para asegurar la disponibilidad de la memoria requerida.

Calcular requerimientos de memoria es un problema desafiante. Evaluarlos de forma cuantitativa es un problema inherentemente difícil siendo indecidible el problema de obtener una cota superior finita para el consumo de memoria [Ghe02].

En [BGY06] se plantea una técnica para aproximar los requerimientos de memoria de un programa Java. Dado un método m con parámetros $\bar{p_m}$ se obtiene una cota superior paramétrica (en términos de $\bar{p_m}$) 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 éste 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.

Ambas técnicas requieren del conocimiento de las distintas configuraciones del stack de llamadas que realiza el método bajo análisis. Es decir, el método requiere conocer los contextos de todas las invocaciones que se generan a partir de él. Esto determina que el análisis no sea composicional. Las ventajas de un análisis composicional son muchas: reutilización de especificaciones, mayor escalabilidad, capacidad de analizar aplicaciones que llaman a métodos no analizables (no se dispone de su código), mejor capacidad para la integración con otras técnicas (por ejemplo [SR05b]), etc.

En la tesis de licenciatura de Rouaux [RG09] se presenta un nuevo enfoque para la técnica antes mencionada, el cual permite abordar el problema de predicción de consumo de forma composicional. Se desarrolla un algoritmo que permite sintetizar un resumen por método que describe los efectos en el *heap*. Dado un método m, el resumen describe la cantidad de objetos requeridos para la ejecución de m. A su vez, especifica los objetos que exceden el tiempo de vida de m permitiendo evaluar qué ocurre con los mismos en los métodos que invocan a m, de aquí proviene la designación de composicional. Una de las características del enfoque mencionado es que utiliza una técnica de análisis para aproximar el tiempo de vida de los objetos diseñada para ser performante, relegando precisión.

En esta tesis abordamos algunos de los problemas abiertos presentados en [RG09], como ser mejorar las cotas paramétricas del consumo de memoria obtenidas en términos de precisión. En [RG09] se utiliza un análisis de tiempo de vida de objetos que modela el *heap* con un grafo no dirigido. Esto trae problemas, por ejemplo, para modelar patrones clásicos como la iteración de colecciones donde el análisis sobreaproxima el tiempo de vida del iterador. La técnica de Points-To definida en [SR05b, SR05a] fue pensada para determinar si un método modifica sus parámetros o algún objeto alcanzable desde ellos y utiliza como modelo del *heap* un grafo dirigido. Esto otorga al análisis mayor información a la hora de modelar las relaciones entre objetos y es la base para mejorar la precisión de la técnica y, en consecuencia, las cotas paramétricas del consumo de memoria obtenidas.

Formalizar la técnica presentada en [RG09] permite generalizar la técnica de análisis de tiempo de vida utilizada, brindando la posibilidad de contar con una herramienta que soporte diferentes instanciaciones de técnicas capaces de aproximar el tiempo de vida de un objeto. Por otro lado, la problemática de analizar programas con llamados polimórficos es una característica que en general es abordada mediante una sobreaproximación. Mejorar las cotas obtenidas en estos casos resulta importante pues los llamados polimórficos son una característica esencial del paradigma de programación orientado a objetos. Esta tesis aborda estos problemas con el objetivo de generalizar la técnica presentada en [RG09] y mejorar las cotas obtenidas en los programas con llamados polimórficos.

1.2. Acerca de este trabajo

Tomando como base los trabajos presentados en [BFGY08] y [RG09], esta tesis presenta un algoritmo composicional capaz de determinar los requerimientos de memoria de programas desarrollados en el lenguaje Java. Al igual que los trabajos mencionados, se asume un modelo de administración de memoria basado en *scopes*. Se define el *scope* de un método m como el conjunto de instrucciones que m ejecuta (esto incluye las líneas de los métodos invocados por m). En este modelo la liberación

de memoria ocurre al finalizar la ejecución de un método. En este esquema sólo es posible controlar lo que ocurre con los objetos creados dentro del *scope* del método ejecutado. Por lo tanto al finalizar la ejecución de m, el modelo permite distinguir la memoria solicitada dentro del *scope* de m que podrá ser liberada, de aquella que deberá ser liberada por un método con un *scope* más grande que incluya a m (i.e. un método que haya invocado a m).

De esta forma es posible asociar el tiempo de vida de los objetos al tiempo de vida de un método. Diremos que un objeto o dentro del *scope* de m es *temporal* si al finalizar la ejecución de m la memoria reservada para su creación puede ser liberada, en otras palabras, si el tiempo de vida de o no supera al de m. Por el contrario será *residual* a m si la memoria reservada para o no pueda ser liberada al finalizar la ejecución de m, es decir, si el tiempo de vida de o excede al de m.

El algoritmo presentado en este trabajo mantiene las bases presentadas en [RG09] para calcular resúmenes de consumo que expresen los requerimientos de memoria. El resumen de un método m con parámetros formales $\bar{p_m}$ y parámetros relevantes $\bar{x_m}$, estará formado por una componente *temporal* y otra *residual* paramétricas en $\bar{x_m}$. Los parámetros relevantes $\bar{x_m}$ son valores enteros que influyen en el consumo, derivables a partir de $\bar{p_m}$. La componente *temporal* representa la cantidad de memoria que será liberada luego de la ejecución de m. La componente *residual* representa la memoria reservada para objetos cuyo tiempo de vida excede al de m.

De esta forma la componente *temporal* de un resumen del método m estará conformada por tres partes:

1. La memoria temporal local: La misma corresponde a los objetos temporales creados en m mediante instrucciones `new` y se calcula *contando* la cantidad de visitas a dichas instrucciones.
2. La memoria temporal necesaria para ejecutar las invocaciones: La misma se calcula tomando el *máximo* entre las componentes *temporales* de los resúmenes de los métodos invocados por m.
3. La memoria temporal producto de las invocaciones: La misma se calcula *acumulando* la memoria especificada como *residual* en los métodos invocados por m que se convierte en *temporal* a m.

En el caso de la componente *residual* estará conformada por dos partes:

1. La memoria residual local: La misma corresponde a los objetos residuales creados en m mediante instrucciones `new` y se calcula *contando* la cantidad de visitas a dichas instrucciones.
2. La memoria residual producto de las invocaciones: La misma se calcula *acumulando* la memoria especificada como *residual* en los métodos invocados por

m que continua siendo *residual* a m.

Notar que la componente *residual* del método m debe proveer la suficiente información para detectar cuánto consumo *residual* se convertirá en *temporal* en el método m' que invoca a m y cuánto continuará siendo *residual* a m'.

De la definición de las componentes se desprende el carácter composicional del cálculo de resúmenes: un resumen se calcula componiendo los resúmenes de los métodos invocados con la información local. Esta característica presenta ventajas tales como: reutilización de especificaciones, mayor escalabilidad, capacidad de analizar programas con métodos no analizables y mayor capacidad de integración a otras técnicas.

Otro aspecto importante que se desprende es el hecho que las expresiones resultantes deben estar en términos de los parámetros del método resumido. Por lo tanto debe existir algún mecanismo que vincule los parámetros del método invocado con los argumentos y vincule estos argumentos con los parámetros del método bajo análisis.

De esta forma, el carácter composicional del análisis requiere la definición de operadores que permitan componer la información del resumen del método bajo análisis con el resumen de un método invocado. En [RG09] se definen los operadores de *conteo*, *acumulación*, *maximización* y *supremo* que operan sobre polinomios que representan el consumo *temporal* y *residual* de un método. El operador de *conteo* permite determinar la cantidad de veces que será visitada una instrucción `new`. Los operadores de *acumulación* y *maximización* trabajan con expresiones a fin de determinar la sumatoria y el máximo de la misma en un cierto rango. La *acumulación* opera sobre el *residual*, mientras que la *maximización* sobre el *temporal*. Además se encargan de realizar la vinculación antes mencionada. El operador de *supremo* se encarga de obtener una cota para el máximo entre dos polinomios. Esta cota es utilizada para obtener la memoria temporal necesaria para ejecutar las invocaciones.

Como se explica en [RG09] el problema de maximización de polinomios que intenta resolver el operador de *supremo* es un problema complejo, siendo en algunos casos imposible determinar una solución exacta. En el trabajo mencionado se aborda este problema dando una cota superior para el máximo y se presentan algunas alternativas para abordar este problema. En este trabajo se extiende el conjunto de operadores con la posibilidad de postergar el momento en que se da el resultado concreto de una operación hasta contar con la información necesaria para que éste sea lo más exacto posible. Si bien esto todavía no resuelve el problema de maximización de polinomios, los resultados obtenidos son más precisos. El algoritmo presentado soporta distintas implementaciones de calculadoras simbólicas que implementen estas operaciones, permitiendo reemplazar la implementación de los operadores, por ejemplo por alguna otra más precisa.

Como se menciono anteriormente, para determinar el tiempo de vida de un objeto, en [RG09] se utiliza una técnica que relega precisión para obtener buena performance. En esta tesis se reemplaza la técnica utilizada en [RG09] por una adaptación de la técnica de Points-To de [SR05b, SR05a] con el objetivo de mejorar la precisión del análisis de tiempo de vida de objetos. La incorporación de esta técnica lleva a utilizar un grafo dirigido con información de lecturas y escrituras como parte del análisis de tiempo de vida de un objeto. Esta forma de modelar el *heap*, al contar con una mayor cantidad de información respecto a las técnicas que utilizan un grafo no dirigido como la utilizada en [RG09], permite resolver patrones que antes no podían ser modelados correctamente. Por ejemplo es posible mejorar la precisión de la iteración de colecciones, dando una mejor aproximación al tiempo de vida de los iteradores.

Al igual que [RG09], el algoritmo desarrollado en este trabajo asume que al momento de calcular el resumen de un método m los resúmenes de los métodos invocados por m ya han sido calculados. Asumir esto limita el análisis de programas recursivos. En caso de necesitar analizar los mismos, es necesario proveer de manera manual un resumen para los nodos que conforman la componente recursiva del *call graph* (i.e. para la componente recursiva del grafo que aproxima los métodos que serán invocados durante la ejecución del programa junto con la relación de orden de ejecución de los mismos). Además, lo asumido permite determinar un orden para calcular los resúmenes de consumo: recorrer de forma bottom-up el *call graph* asociado al programa.

En este trabajo se propone:

1. Un algoritmo de análisis composicional que soporte instanciaciones de diferentes técnicas capaces de aproximar el tiempo de vida de un objeto y distintas instanciaciones de la calculadora que opera con expresiones de consumo, y ataque el problema de polimorfismo con el objetivo de mejorar las cotas en los casos polimórficos.

2. Reemplazar el análisis que aproxima el tiempo de vida de los objetos presentado en [RG09] para mejorar las cotas obtenidas en términos de precisión.

3. Evaluar la técnica desarrollada utilizando diferentes programas, uno de ellos de tamaño medio que involucra 36 clases.

A lo largo de este trabajo cada uno de las propuestas enunciadas es abordada proveyendo técnicas que luego son implementadas. La propuesta 1 es abordada mediante un nuevo algoritmo presentado y desarrollado en el Capítulo 4. El mismo toma como base al algoritmo presentado en [RG09] adaptándolo para soportar distintos análisis que aproximen el tiempo de vida de los objetos y distintas implementaciones de los operadores que componen la calculadora simbólica, y abordar el problema de

polimorfismo con el objetivo de mejorar las cotas obtenidas para casos polimórficos. Para esto se generaliza el problema y se desprenden nuevos conceptos explicados en las diferentes secciones como ser particiones, evaluación lazy, etc.

La propuesta 2 es abordada al utilizar el análisis propuesto en [SR05b, SR05a] como implementación del análisis de tiempo de vida de objetos. En el Capítulo 5 se dará una explicación detallada del mismo.

La propuesta 3 es abordada en el Capítulo 8 definiendo una serie de casos de estudio en donde se infieren los resúmenes de todos los métodos que componen un programa utilizando la herramienta implementada a lo largo de este trabajo y se los compara con los resultados de la herramienta presentada en [RG09]. Para esto, se utiliza programas del *benchmark* JOlden [CM01] y la aplicación de tamaño medio para JAVA *JLayer*.

En el Capítulo 6 se presenta el conjunto de operaciones desarrollado y necesario para abordar las propuestas realizadas. A modo de síntesis, se presenta una interfaz que permite definir expresiones paramétricas las cuales serán implementadas usando la calculadora `iscc` basada en la librería Barvinok [Ver07, Ver10]. Esta librería permite la manipulación de expresiones paramétricas y cuenta con diferentes operaciones las cuales son explicadas con mayor detalle en la Sección 7.3. Este enfoque pretende a su vez determinar el costo de trabajar con la librería vía línea de comandos (ver Sección 7.3).

1.3. Visión General

Como fue mencionado en la sección anterior, el objetivo de este trabajo es desarrollar un algoritmo de análisis composicional capaz de soportar diferentes técnicas de estimación de tiempo de vida de un objeto y diferentes implementaciones de una calculadora simbólica. A su vez, se pretende desarrollar una herramienta que permita realizar una prueba conceptual utilizando el lenguaje Java, esperando mejorar los resultados obtenidos en trabajos anteriores.

Para el cálculo de consumo se asume un esquema de administración de memoria basado en *scopes* (ver Sección 2.2). En este esquema, al finalizar la ejecución de un método m sólo pueden ser recolectados los objetos creados por m o por un método al que m invoca.

Nuestro prototipo permite sintetizar estimadores paramétricos que aproximan la cantidad total de objetos necesarios para ejecutar de manera segura un método (ver Secciones 2.3 y 4.1). 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 (EA): Aproxima de forma automática el

tiempo de vida de los objetos (ver Sección 2.2.2, 6.3.3 y 7.2).

- Proveedor de invariantes locales: Es responsable de proveer los invariantes requeridos por las técnicas de predicción de memoria (ver Sección 7.1).
- CT: Es capaz de contar la cantidad de visitas a una instrucción de un programa (ver Secciones 4.2 y 7.3).
- SA: Es capaz de operar con expresiones paramétricas (ver Secciones 4.2 y 7.3).
- Calculadora de expresiones paramétricas: Resuelve las operaciones requeridas por el CT y el SA (ver Sección 7.3).
- Analizador de memoria dinámica: Infiere estimadores paramétricos de la cantidad de objetos requeridos por un método (ver Secciones 6.3.3 y 7.4).

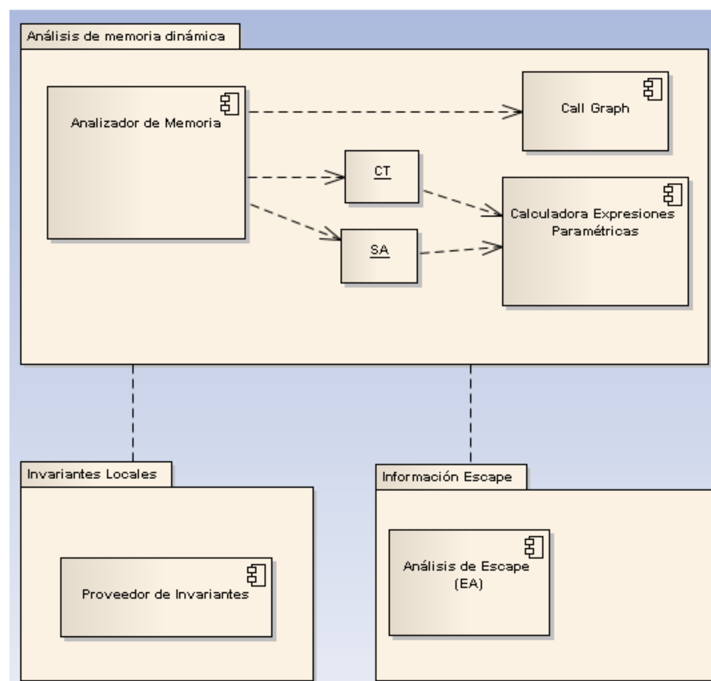


Figura 1.1: Componentes centrales que integran la solución propuesta.

El *Analizador de memoria dinámica* a partir de la información obtenida por el análisis de escape (EA) y usando el CT y el SA, sintetiza expresiones paramétricas que acotan de manera conservadora la cantidad de objetos requeridos por la ejecución de un método. Este componente junto con la implementación del análisis de escape (para aproximar el tiempo de vida de los objetos) presenta el mayor esfuerzo de esta tesis.

La técnica desarrollada propone un algoritmo de inferencia de resúmenes por método considerando el árbol de llamadas. Es decir, para obtener el resumen de un método m se utiliza el conjunto de resúmenes 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 se debe calcular primero el resumen de m'. Para esto, el algoritmo propuesto recorre el árbol de llamadas de forma bottom-up, requiriendo operaciones de suma y maximización sobre los estimadores de consumo ya calculados. Estas operaciones son resueltas por el SA a través de la *Calculadora de expresiones paramétricas*.

La *Calculadora de expresiones paramétricas* implementa las operaciones necesarias usando como base la calculadora `iscc` . En conjunto con el componente de *Invariantes Locales* permite implementar las herramientas CT y SA necesarias para el análisis de consumo de memoria. El hecho que sea un componente independiente permite reemplazar la implementación basada en `iscc` por otra.

El componente que implementa el *Análisis de escape* es resuelto mediante una adaptación del análisis propuesto en [SR05b, SR05a].

El *Proveedor de invariantes locales* es el modulo más simple puesto que asume que los invariantes serán provistos mediante un XML. Si bien esto no aborda el problema de generación de invariantes, deja las puertas abiertas a integrar la herramienta con cualquier otra existente que genere los mismos y pueda escribirlos en el formato esperado.

1.4. Resumen de contribuciones

Esta tesis realiza tres contribuciones. Primero, propone un algoritmo de análisis composicional que generaliza el análisis presentado en [RG09] para soportar instanciaciones de diferentes técnicas capaces de aproximar el tiempo de vida de un objeto y de diferentes calculadoras simbólicas, y aborda el problema de polimorfismo dentro del análisis con el objetivo de mejorar las cotas obtenidas como resultado.

La segunda contribución es una instanciación del algoritmo presentado que utiliza un análisis para aproximar el tiempo de vida de los objetos basado en [SR05b, SR05a] y una calculadora simbólica basada en la librería Barvinok [Ver10]. Se aprovechan las características de la librería maximizando el uso de las operaciones que la misma provee para mejorar los resultados obtenidos en términos de precisión. Además, se presentan distintas alternativas para resolver problemas surgidos en los casos polimórficos a partir de la instanciación elegida.

Finalmente se presenta una evaluación de la técnica desarrollada utilizando diferentes programas, uno de ellos de tamaño medio que involucra 36 clases.

1.5. Estructura

En el Capítulo 2 se presentan conceptos utilizados a lo largo del desarrollo de esta tesis.

En el Capítulo 3 se presenta una visión general del enfoque que fue utilizado para inferir requerimientos dinámicos de memoria usando una serie de ejemplos simples. Se muestra de manera informal cómo la información cuantitativa sobre el consumo de memoria es calculada utilizando la información local del método, y cómo las llamadas a otros métodos son procesadas usando los resúmenes de los métodos.

El Capítulo 4 presenta un algoritmo composicional para construir resúmenes de consumo de memoria capaz de soportar diferentes técnicas para el cálculo del tiempo de vida de un objeto y diferentes implementaciones de calculadoras simbólicas.

En el Capítulo 5 se presenta el análisis para aproximar el tiempo de vida de los objetos utilizado para la instanciación del algoritmo realizada inspirado en [SR05b, SR05a], adaptado con la posibilidad de ser k-sensitivo.

En el Capítulo 6 se presenta una instanciación del algoritmo descripto y el análisis de distintas alternativas para abordar el polimorfismo y resolver los operadores necesarios.

En el Capítulo 7 se presentan los detalles más relevantes relacionados a la implementación de la herramienta desarrollada para el estudio de consumo de memoria.

Los Capítulos 8 y 9 presentan los experimentos realizados, conclusiones, limitaciones, trabajos relacionados y trabajos a futuro.

Finalmente el apéndice A presenta documentación técnica de la herramienta desarrollada.

Capítulo 2

Conceptos preliminares

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

2.1. Notación para programas

Definición Un programa es un conjunto $\{m_0, m_1, \ldots\}$ de métodos. Llamamos *Methods* a este conjunto. □

Definición Un método m posee una lista $\bar{p_m}$ de parámetros, un conjunto de variables locales y un cuerpo que consiste en una lista de instrucciones. Cada instrucción podrá ser identificada unívocamente por un label lb. □

Definición Denominamos *Labels* al conjunto de todos los labels lb existentes en el programa. □

Definición Dado un método m se define la función $params(m)$ que retorna la lista de parámetros del método m. □

Definición Dado un método m se define la función $body(m)$ que retorna la lista de instrucciones del método m. □

Definición Las instrucciones de un método tienen la forma $lb : instruccion$. En particular se distinguen las siguientes instrucciones:

- lb : *new* T: creación de un objeto de tipo T, el cual podría ser un arreglo.
- lb : *call* $m(\bar{args})$: invocación al método m (con parámetros formales p_1,...,p_n) con argumentos a_1,...,a_n.

Se reserva el uso de esta notación para la formalización de conceptos, mientras que para facilitar la comprensión de los ejemplos se utilizará notación tipo *Java*.

Definición Denominamos *call graph* al grafo dirigido en el cual cada nodo representa un método y cada eje de tipo (f, g) indica que el método f invoca al método g. Lo notamos como CG y permite aproximar, en tiempo de compilación, los métodos que serán invocados durante la ejecución de un programa junto con una relación de orden de ejecución de los mismos. El mismo debe ser capaz de resolver la operación $getCallees$ que dado el label de una instrucción de tipo lb : *call* $m(\bar{args})$ retorna el conjunto de métodos candidatos a ser ejecutados producto de la invocación. Puede existir más de un candidato en caso que la invocación sea virtual (e.g. se invoca un método definido en un interfaz).

2.2. Organización de memoria predecible

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 utilizada para sistemas embebidos de tiempo real. La razón principal es que resulta extremadamente difícil predecir los momentos en que la memoria será liberada de forma efectiva.

Una solución posible al problema de liberación de memoria dinámica es asignar objetos a *scopes* [PFHV04], como se realiza en [RG09]. Se define el *scope* como el conjunto de instrucciones que m ejecuta (esto incluye las líneas de los métodos invocados por m). En este esquema, al finalizar la ejecución de un método m sólo es posible liberar memoria reservada en el *scope* de m. Por lo tanto es posible distinguir la memoria solicitada dentro del *scope* de m que podrá ser liberada, de aquella que deberá ser liberada por un método con un *scope* más grande que incluya a m (i.e. un método que haya invocado a m). El primer tipo de memoria será denominada *temporal* y el segundo tipo *residual*.

Para lograr una administración de memoria predecible, este esquema requiere de herramientas que permitan asignar, en tiempo de compilación, el *scope* en el que la memoria reservada para la creación de un objeto podrá ser liberada. Esto puede ser determinado por cualquier técnica capaz de aproximar el tiempo de vida de un objeto.

2.2.1. Tiempo de vida de un objeto

El tiempo de vida de un objeto se define como el período que ocurre desde la creación del mismo hasta que ya no es referenciado por otro objeto. No es posible determinar en tiempo de compilación el momento exacto en que esto ocurre [Bla03]. Una posible solución a este problema consiste en asociar el tiempo de vida de un objeto al tiempo de vida de un método. Esto simplifica el problema pues el tiempo de vida de un método puede ser aproximado en tiempo de compilación. De esta forma, el tiempo de vida puede ser considerado como el período que ocurre desde la creación del objeto hasta la finalización de la ejecución del método en donde el objeto deja de ser referenciado.

Asociar el tiempo de vida de un objeto o al de un método m equivale a asociar el objeto al *scope* de m y es lo que permite predecir cuándo será liberada la memoria reservada para o.

2.2.2. Análisis de escape

El objetivo del análisis de escape es aproximar el tiempo de vida de un objeto, es decir aproximar el método en donde el objeto deja de ser referenciado. 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 stack allocation. 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.

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

Definición Un objeto es *capturado* por un método m cuando su tiempo de vida no es mayor al tiempo de vida de m. □

Ejemplo Consideremos a modo de ejemplo el Código 2.1 donde el método `shuffle` retorna una lista de enteros aleatorios.

```
public class EscapeExample {

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

Listing 2.1: Ejemplo de análisis de escape.

Es posible determinar en tiempo de compilación que el objeto `now` creado en la línea 1 y el objeto `random` creado en la línea 2, no escapan al método `shuffle` y en consecuencia pueden ser almacenados en la pila. Por otro lado el objeto *retList* creado en la línea 3 es retornado, por lo tanto el tiempo de vida de este objeto es mayor al del método y en consecuencia no puede ser alocado en el stack del método. Lo mismo ocurre con todos los objetos creados en la línea 5.

Como se mencionó anteriormente, nos interesa determinar de forma conservadora cuándo un objeto escapa o es capturado por un método m a fin de determinar si es posible asociar al objeto el *scope* de m y por lo tanto catalogarlo como miembro del temporal o residual del método. Para esto se utilizará una adaptación del análisis de escape presentado en [SR05b, SR05a] que utiliza como base un Points-To. La técnica de Points-To permite detectar las posiciones de memoria a las que hace referencia un puntero.

2.3. Resúmenes de consumo de memoria

Un resumen de consumo de memoria debe proveer información acerca de la cantidad de memoria que un método requiere para poder ser ejecutado. Dado que se asume un manejo de memoria automático basado es *scopes*, para poder cumplir con el carácter composicional se define que el resumen incluya detalles del tiempo de vida de los objetos sobre los que predica. Esto permite modelar tanto la asignación como la liberación de objetos en memoria.

Para cumplir con estos requisitos y facilitar el razonamiento composicional, se toman las siguientes decisiones respecto al diseño de los resúmenes de memoria:

- Dado un método m, se realiza una partición de los objetos con los que m trabaja basada en su tiempo de vida, de la siguiente manera:

    1. Objetos *temporales*: representan aquellos objetos creados en el *scope* de m que son capturados por m.

2. Objetos *residuales*: representan aquellos objetos creados en el *scope* de m que escapan m.

- Los objetos residuales estarán agrupados en *particiones*. Esto permite representar conjuntos de objetos con distinto tiempo de vida. La cantidad de particiones de un resumen dependerá del nivel de granularidad deseado para el análisis (puede variar en función de la precisión del análisis de escape utilizado).

- El resumen especifica únicamente información sobre los objetos creados dentro del *scope* del método resumido.

- La información de consumo de un método m con parámetros formales $\bar{p_m}$ será expresada en términos de parámetros relevantes ($\bar{x_m}$) al consumo de m. Los parámetros relevantes pueden diferir de los parámetros formales de m, pero debe existir una relación entre ambos tipos de parámetros (i.e. los parámetros relevantes podrían derivarse de los formales). Esto permite, por ejemplo, utilizar parámetros relevantes derivados de parámetros formales no enteros, como ser el tamaño del parámetro formal lista. Además es posible introducir manualmente parámetros relevantes cuya relación con los parámetros formales no es inferible de manera trivial, como ser un parámetro relevante que exprese el máximo tamaño de las listas contenidas en una lista (i.e. el parámetro formal es una lista de listas). Por lo tanto especificar el consumo en términos de parámetros relevantes permite mejorar las cotas en escenarios donde el programa analizado es suficientemente complejo.

Definición Una *expresión paramétrica* $e \in E : \bar{X} \mapsto \bar{K} \mapsto N$, es una función de valuaciones de parámetros relevantes e instanciaciones de tipos (e.g. sus tamaños) en números naturales. Llamamos $\mathbb{E}$ al conjunto de las expresiones paramétricas.

Ejemplo Sean n y m parámetros enteros de una expresión, $3*n*integer+m*object$ es una expresión paramétrica en la cual *object* e *integer* representan el tamaño de los tipos *Object* e *Integer* respectivamente.

Definición Un reticulado es un conjunto parcialmente ordenado en el cual dos elementos cualesquiera del mismo poseen un supremo (la menor cota superior de los elementos, llamada el *join* de ambos) y un ínfimo (la mayor cota inferior, llamada el *meet* de ambos).

Definición Un semi-reticulado superior es un conjunto parcialmente ordenado en el cual dos elementos cualesquiera del mismo poseen un supremo.

Definición Un semi-reticulado superior de expresiones paramétricas está definido por $\langle E, 0, \infty, \leq, join\rangle$ donde 0 cumple la función del ínfimo de las expresiones paramétricas, ∞ la del supremo de toda expresión, $\leq$ la de comparación de expresiones y *join* la de *join*.

Definición Una *partición* del *heap*[1] es un conjunto de objetos alcanzables a partir de una referencia existente en el método que está siendo resumido. Una partición define la mínima unidad de información sobre tiempos de vida que es provista por un resumen. HP denota el conjunto de posibles particiones del *heap*.

A los fines de los ejemplos se utilizará un *nombre* de partición para referirse al objeto representado por ese nombre y todos los objetos alcanzables desde el mismo. Por ejemplo, *cell* podría referirse a un objeto *cell* (que representa un eslabón de una lista enlazada) y los objetos alcanzables desde el mismo como ser *cell.data*, *cell.next* y *cell.next.next*. En general, se utilizará *ret* para referirse a la partición relacionada a los objetos retornados por un método y todos los objetos alcanzables desde ellos.

Ejemplo Consideremos el siguiente método:

```
1  Object  m1(C p1, int n) {
2      p1.f.g = new Integer[n];
3      p1.f.h = new Integer();
4      return new Object();
5  }
```

Listing 2.2: Método con varios objetos que escapan.

Dado que el consumo de memoria del método depende del parámetro n, el mismo es un parámetro relevante del método. Por lo tanto, un posible resumen para el método es: $S.Tmp = 0$, $S.Rsd(ret) = object$, $S.Rsd(p1) = integer * n + integer$. Una versión más refinada podría ser: $S.Tmp = 0$, $S.Rsd(ret) = object$, $S.Rsd(p1.f.g) = integer * n$, $S.Rsd(p1.f.h) = integer$.

Es importante notar que en este caso los parámetros relevantes coinciden con los parámetros formales del método. Sin embargo, esto podría no ocurrir en todos los casos.

Ejemplo Consideremos el método `Tree copy(int n)` que hace una copia de los primeros n niveles de un árbol. Los parámetros relevantes del método `copy` pueden ser n, $this.size()$ o cualquier otro valor entero que pueda ser derivado de esos parámetros o de algún campo estático. Incluso puede incluirse un parámetro relevante h que represente la altura del árbol si es que la misma resultase relevante para dar las expresiones de consumo del método. Como se mencionó anteriormente, las ex-

[1] Partición se refiere a un fragmento del *heap* y no al concepto clásico de partición de conjuntos

presiones de consumo del método podrían depender de parámetros relevantes del método derivados de algunos de los parámetros formales del mismo.

Definición Dado un método m con parámetros formales $\bar{p_m}$ se define su *resumen de consumo de memoria* como la tupla $S_m = \langle \bar{p}, \bar{x}, \bar{\gamma}, \bar{k}, Tmp, Rsd \rangle$ donde:

- $\bar{p} : \bar{P}$ es la lista $\bar{p_m} = p_1, ..., p_n$ de parámetros formales del método.
- $\bar{x} : \bar{X}$ es la lista $\bar{x_m} = x_1, ..., x_k$ de parámetros relevantes, donde cada parámetro representa a un valor entero. Se considera parámetro relevante a toda expresión derivable de $\bar{p_m}$ que influye en el consumo del método m.
- $\bar{\gamma} : \bar{\Gamma}$ es la lista $\bar{\gamma_m} = \gamma_1, ..., \gamma_k$ de funciones las cuales relacionan los parámetros formales del método con los parámetros relevantes del mismo, es decir, $\gamma_i : \bar{P} \mapsto X$ representa la derivación a realizar para obtener el parámetro relevante x_i a partir de los parámetros formales del método m.
- $\bar{k} : \bar{K}$ es la lista de tipos correspondiente a los objetos que aparecen en las expresiones.
- $Tmp : \mathbb{E}[\bar{X}][\bar{K}]$ es una expresión que representa el consumo de los objetos temporales.
- $Rsd : \mathsf{HP} \mapsto \mathbb{E}[\bar{X}][\bar{K}]$ es una función parcial que dada una partición del *heap* retorna una expresión paramétrica que representa el consumo de los objetos incluidos en esa partición. Notamos al dominio de la función como $Dom(S_m.Rsd)$.

Se asume que para resumir un método m tanto los parámetros relevantes ($\bar{x_m}$) como el mapping entre parámetros formales y relevantes ($\bar{\gamma_m}$) son dados. No forma parte de este trabajo investigar la generación automática de ambas componentes.

Ejemplo Consideremos nuevamente el Código de ejemplo 2.2. El resumen del método `m1` puede definirse como:

$$\langle [p1, n], [n], [\gamma_{m1}], [integer, object], 0, Rsd \rangle$$

donde $\gamma_{m1}(p1, n) = n$, $Rsd(ret) = object$ y $Rsd(p1) = integer * n + integer$.

Ejemplo 2 Consideremos nuevamente el método `Tree copy(int n)` que hace una copia de los primeros n niveles de un árbol. Su consumo podría ser resumido como:

$$\langle [this, n], [nodesUpToLevel_n], [\gamma_{nodesUpToLevel}], [node], 0, Rsd \rangle$$

donde $\gamma_{nodesUpToLevel}(n, this) = nodesUpToLevel_n$, $Rsd(ret) = node * nodesUpToLevel_n$, $node$ es el tamaño de un nodo del árbol y $nodesUpToLevel_n$ representa la cantidad de nodos que contiene el árbol en los primero n niveles.

En este caso se puede ver cómo la derivación de los parámetros relevantes a partir de los parámetros formales no siempre resulta trivial.

Por razones de simplicidad, se asumirá que todos los objetos tienen tamaño 1, es decir $\forall k \in \bar{k}$, el tamaño de $k = 1$. Por lo tanto se dejará las expresiones únicamente en términos de una lista de parámetros, en donde $\mathbb{E}[\bar{X}]$ denotará expresiones que esperan una lista de parámetros $[x_1 : Integer, \ldots, x_k : Integer]$. En consecuencia en los ejemplos anteriores *node*, *integer* y *object* tendrán un valor de 1.

Definición Dado una lista de mappings $\bar{\gamma_m}$ y una lista de parámetros formales $\bar{p_m}$ de un método m, se define el azucar sintáctico $\bar{\gamma_m}(\bar{p_m}) = [\gamma_i(\bar{p_m}) \mid \gamma_i \in \bar{\gamma_m} \wedge i = 1...|\bar{\gamma_m}|]$ el cual representa la lista de parámetros relevantes que se obtiene producto de evaluar cada función perteneciente a $\bar{\gamma_m}$.

Definición Sea un método m, se define el estimador paramétrico $MemReq_m$ el cual aproxima el mayor número de objetos vivos durante la ejecución de m y los métodos invocados por m. Dado el resumen S_m de m, $MemReq_m$ puede ser calculado como

$$S_m.Tmp + \sum_{hp_m \in Dom(S_m.Rsd)} S_m.Rsd(hp_m)$$

Ejemplo Consideremos el Código de ejemplo 2.2. Siendo que $\forall k \in \bar{k}$, el tamaño de $k = 1$ y por lo tanto $integer = object = 1$, el valor para el estimador definido es $MemReq_{m1} = n + 2$.

En el capítulo siguiente se presenta una visión general del enfoque utilizado para inferir requerimientos de memoria dinámica usando una serie de ejemplos simples.

Capítulo 3

Ejemplos motivacionales

En este capítulo se presenta una visión general del enfoque utilizado para inferir requerimientos de memoria dinámica usando una serie de ejemplos simples. Se muestra de manera informal cómo la información cuantitativa sobre el consumo de memoria es calculada utilizando la información local del método y los resúmenes de los métodos invocados. Se utilizarán los conceptos de partición temporal y residual y sus efectos sobre el cálculo general.

Por simplicidad, de aquí en adelante se considera el número de objetos en memoria en vez del espacio que los mismos ocupan (esta información puede ser calculada agrupando los objetos por su tipo). Los arreglos de dimensión d son considerados como d objetos de tipo $T[]$ (referencia a T).

Además, para simplificar los ejemplos sólo se brinda las componentes Tmp y Rsd de los resúmenes y se deja implícita la componente $\bar{\gamma}$ de los resúmenes. La misma podrá ser deducida de la notación utilizada para los parámetros relevantes. De esta forma, si un método tiene un parámetro formal l de tipo lista y en el resumen figura una expresión de tipo $l.size$, queda implícita la función $\gamma_{l.size}(l) = l.size$. Lo mismo ocurre con los parámetros v de tipo arreglo y el parámetro formal $v.length$. Se mencionará la función de mapping utilizada en el caso que la misma no se deduzca trivialmente de la notación utilizada.

Ejemplo 1 Consideremos el método `multiply` presentado en el Código 3.1 que crea un nuevo arreglo multiplicando cada elemento del arreglo de entrada por un factor entero. Para inferir su consumo, se necesita acotar la cantidad de veces que cada instrucción `new` (creación de objetos) es visitada. Notar que para analizar la cantidad de visitas a las instrucciones de las líneas 3 y 4, se necesita una técnica capaz de acotar el número de veces que el ciclo es ejecutado (**conteo**) (ver Sección 6.3.1).

```
0 Integer[] multiply(Integer[] v,int f){
1   Integer[] res = new Integer[v.length];
2   for(int i=0;i<v.length;i++) {
3     Integer factor = new Integer(f);
4     res[i] = new Integer(v[i] * factor);
5   }
6   return res;
7 }
```

Listing 3.1: Ejemplo 1.

Existen distintas técnicas que tratan con este problema. Por ejemplo [BGY06] obtiene una expresión paramétrica del número de visitas contando las soluciones de un invariante de programa que describe el espacio de iteración. Existen otros enfoques como [AGGZ09, GZ10] que se basan en funciones variantes y el análisis de patrones de iteración.

El método `multiply` crea en la línea 1 un arreglo de $v.length$ objetos (de tipo `Integer[]`), $v.length$ objetos de tipo `Integer` en la línea 3, y $v.length$ objetos de tipo `Integer` en la línea 4. Tanto el arreglo como los objetos asignados en la línea 4 viven más tiempo que el método, es decir que escapan, ya que son retornados al método llamador. Por lo tanto, se ubica a esos objetos en la partición residual del consumo del método, indicando que escapan a través del valor de retorno. Por otro lado, los objetos creados en la línea 3 son sólo temporales, lo que significa que ninguno de ellos es requerido cuando el método finaliza su ejecución. Por lo tanto, se ubica a esos objetos en la partición temporal del consumo del método.

Luego, el consumo de `multiply` puede ser resumido como `multiply`.$Tmp = v.length$ y `multiply`.$Rsd(ret) = 2 * v.length$. Notar que se realizo un vínculo entre los parámetros formales v y f del método analizado con el parámetro relevante $v.length$ para referirse al largo del vector.

3.1. Analizando llamadas

Para poder acotar la memoria dinámica requerida por un método m, se necesita también considerar todas las llamadas a otros métodos realizadas por m. La idea es confiar en el resumen de cada método llamado e introducir reglas de composición para reflejar el efecto de esas ejecuciones en el método llamador.

Ejemplo 2 Consideremos el método `testMultiply` presentado en el Código 3.2 el cual no crea localmente ningún objeto, pero llama dos veces a `multiply` el cual crea objetos.

```
0 void testMultiply(Integer[] v1, Integer[] v2) {
1   Integer[] m1 = multiply(v1,2);
2   Integer[] m2 = multiply(v2,5);
3 }
```

Listing 3.2: Ejemplo 2.

Considerando el resumen de `multiply` y la definición de `testMultiply` se puede observar que:

- Ningún objeto creado por las llamadas de `testMultiply` es requerido luego de la ejecución del mismo.
- La primer llamada a `multiply` requiere $v1.length$ objetos temporales, mientras que la segunda requiere $v2.length$. Dado que esos objetos pueden ser recolectados al finalizar cada llamada se debe considerar sólo la expresión máxima, es decir $max\{v1.length, v2.length\}$.
- Cada partición residual generada por una llamada a `multiply` es temporal en el llamador. La primera de ellas genera $2 * v1.length$ objetos, mientras que la segunda $2 * v2.length$. Dado que esos objetos pueden ser recolectados recién al finalizar la ejecución del llamador se debe considerar la suma de las expresiones, es decir $2 * v1.length + 2 * v2.length$.
- Luego, `testMultiply` puede ser resumido como `testMultiply`$.Tmp = 2 * v1.length + 2 * v2.length + max\{v1.length, v2.length\}$ y `testMultiply`$.Rsd = \emptyset$

Acumulación y Supremo En general, por cada llamada hecha por un método bajo análisis hay que tener en cuenta los objetos temporales y residuales que cada una requiere. Dado que los objetos temporales pueden ser recolectados luego de la ejecución de cada llamada, alcanza con considerar la máxima expresión (**supremo**) (ver Sección 6.3.2). El ejemplo anterior muestra que alcanza con considerar $max\{v1.length, v2.length\}$ objetos temporales. Notar que se debe estar preparado para poder computar el supremo entre expresiones de consumo (ver Sección 6.3.2). Respecto a los objetos residuales de cada llamada, dado que los mismos podrían llegar a ser recolectados luego de la ejecución del método bajo análisis, se necesita acumularlos considerándolos temporales o residuales según sean o no recolectados luego de la ejecución del método analizado (**acumulación**) (ver Sección 6.3.2). En el ejemplo anterior son recolectados por lo que se acumula $2*v1.length+2*v2.length$ objetos temporales.

Ejemplo 3 Consideremos el Código 3.3 que presenta una generalización de `multiply` introduciendo una operación de `map` entre listas. En vez de multiplicar el número, se utiliza un transformador para multiplicar un número entero con un número elegido al azar. Se asume que el método `add` puede ser resumido como $\texttt{add}.Tmp = 0$ y $\texttt{add}.Rsd(ret) = 1$ y que los constructores de las diferentes clases son analizados pero no generan consumo.

```
0 List map(List list, Trans transform) {
1   List res = new List();
2   for (Iterator it = list.iterator(); it.hasNext();) {
3     Object o = transform.apply(it.next());
4     res.add(o);
5   }
6   return res;
7 }
8 class Trans {
9  Object apply(Object object) {
10    Date date = new Date();
11    Random random = new Random(date.getTime());
12    int value = ((Integer) object).intValue();
13    return new Integer( value * random.nextInt());
14  }
15 }
```

Listing 3.3: Ejemplo 3.

Lo primero que se debe notar para el método `apply` es que tanto `date` como `random` son objetos temporales, mientras que el `new Integer` creado en la línea 13 es retornado convirtiéndose en residual. Dado que ninguna llamada genera solicitudes de memoria dinámica, el consumo de `apply` es $\texttt{apply}.Tmp = 2$ y $\texttt{apply}.Rsd(ret) = 1$.

Para calcular el consumo de `map` primero se analizan todos los objetos locales creados y luego cada una de las llamadas. La `List` creada por `map` es retornada por lo que debe ser considerada como residual. La llamada a `iterator` genera un nuevo objeto que es capturado por `map` aumentando su partición temporal en 1. Las llamadas de las líneas 3 y 4 se producen dentro de un ciclo por lo que se tiene que considerar el efecto de dichas llamadas a lo largo de la iteración.

Cada llamada a `apply` producirá 2 objetos temporales y 1 residual. De manera análoga al ejemplo anterior, para los objetos temporales se tiene que tener en cuenta sólo el peor caso de las llamadas a lo largo de todas las iteraciones, donde cada una de ellas es indicada mediante su número (i). Notar que se realiza un vínculo entre los parámetros formales $list$ y $transform$ del método `map` con el parámetro relevante $list.size$ para referirse al tamaño de la lista $list$.

$$\max_{i=1}^{list.size} (\texttt{apply}.Tmp)(i) = \max_{i=1}^{list.size} 2 = 2$$

Para este ejemplo la operación resulta simple ya que si bien cada iteración llama al mismo método podría depender del número de iteración i. Además en este caso la

llamada requiere de una cantidad constante de objetos temporales, pero esto puede implicar un problema de maximización sobre expresiones simbólicas complejas (ver ejemplo 5).

Por otro lado, el objeto retornado por `apply` se inserta en la lista retornada por `map` por lo que escapa y se debe acumular este objeto residual a lo largo de la iteración. Lo mismo ocurre para `add` que genera el nodo requerido para almacenar un nuevo valor como objeto residual. Sea $Rsd(newList)$ el consumo residual aportado por el punto de creación de la lista utilizada como retorno, se debe realizar el siguiente cálculo para obtener el consumo residual de `map`.

$$Rsd(newList) + \sum_{i=1}^{list.size} (\texttt{apply}.Rsd(ret)(i) + \texttt{add}.Rsd(ret)(i)) = 1 + 2 * list.size$$

Luego, el consumo de `map` puede ser resumido como $\texttt{map}.Tmp = 3$ y $\texttt{map}.Rsd(ret) = 1 + 2 * list.size$.

Maximización y Acumulación Como se mostró en el ejemplo 3, todos los objetos temporales de un método m pueden ser recolectados luego de su ejecución. Dado que se está sobreaproximando se necesita decidir qué llamada a m va a necesitar la mayor cantidad de memoria. Más aún, el método m puede ser llamado dentro de un ciclo por lo que se necesita caracterizar la cantidad posible de memoria que esa llamada puede implicar en cada iteración. Notar que en el ejemplo anterior las llamadas devolvían un valor constante, pero éstas podrían devolver expresiones simbólicas de mayor complejidad. Por lo tanto, para poder calcular esa cota se necesita maximizar expresiones simbólicas sobre un espacio de iteración (**maximización**) (ver Sección 6.3.2). La regla de acumulación es necesaria a la hora de considerar las particiones residuales de los métodos llamados. Esto implica una suma del tamaño de cada partición residual sobre el espacio de iteración. En el ejemplo 3, el espacio de iteración es definido por el tamaño de la lista iterada. Por lo tanto, para poder calcular esa expresión se necesita acumular expresiones simbólicas sobre un espacio de iteración (**acumulación**) (ver Sección 6.3.2)

Ejemplo 4 Consideremos como nuevo ejemplo el Código 3.4 donde se agrega una nueva clase transformadora. `Trans2` sobrescribe al método `apply` retornando un arreglo de dos enteros que contiene una copia del número original y uno nuevo generado aleatoriamente.

```
0 class Trans2 extends Trans {
1  Object apply(Object object) {
2    Date date = new Date();
3    Random random = new Random(date.getTime());
4    int value = ((Integer) object).intValue()
5    return new Integer[]{
6  new Integer(value),
7  new Integer(value * random.nextInt())};
8  }
9 }
```

Listing 3.4: Ejemplo 4.

`Trans2.apply` requiere 2 objetos temporales (`date` y `random`) y retorna un `Integer[]` de largo 2 que contiene 2 nuevos enteros (`new Integer`). Luego, se puede resumir el consumo de `Trans2.apply` como `Trans2.apply`.$Tmp = 2$ y `Trans2.apply`.$Rsd(ret) = 4$.

La consecuencia de agregar dicha clase es que mientras se analiza `map` no es posible inferir estáticamente que instancia de `apply` será ejecutada.

Más aún, las implementaciones tienen expresiones diferentes para la partición residual, por lo que para obtener una cota superior de la cantidad de memoria requerida para ejecutar `map` se necesita decidir qué implementación producirá el peor caso. Por lo tanto, la partición residual de `map` es:

$$Rsd(newList) + \sum_{i=1}^{list.size} (\texttt{add}.Rsd(ret)(i) +$$
$$max(\texttt{Trans.apply}.Rsd(ret)(i), \texttt{Trans2.apply}.Rsd(ret)(i))) =$$

$$= 1 + \sum_{i=1}^{list.size} (1 + max(1, 4)) = 1 + list.size * (1 + 4) = 1 + list.size * 5$$

Luego, el consumo de `map` puede ser resumido como `map`.$Tmp = 3$ y `map`.$Rsd(ret) = 1 + 5 * list.size$.

Notar que el número de objetos temporales es el mismo para cada implementación pero puede llegar a ser diferente. En tal caso se debe considerar el peor caso tanto para la parte temporal como la residual.

Polimorfismo El ejemplo anterior muestra cómo las llamadas virtuales pueden ser manejadas considerando la máxima partición residual entre dos implementaciones diferentes. En general, para cualquier llamada virtual se necesita obtener una cota

superior de la cantidad de objetos temporales y residuales entre todas las posibles implementaciones. Se puede confiar en el grafo de llamadas para detectar dichas implementaciones. Cuando se produce una llamada virtual dentro de un ciclo puede ocurrir que diferentes iteraciones invoquen diferentes implementaciones. En ese caso se necesita de una regla más compleja, ya que obtener una cota superior puede requerir sumar diferentes máximos a lo largo del espacio de iteración.

Ejemplo 5 Se presenta un ejemplo final para ilustrar la metodología completa.

```
0 List copy(List list) {
1   List res = new List();
2   for (Iterator it = list.iterator(); it.hasNext();)
3     res.add(it.next());
4   return res;
5 }
6
7 List safeMap(List list,Trans transform) {
8   List cp = copy(list);
9   return map(cp, transform);
10 }
11
12 List test(List<List> list, Trans transform){
13   List res = new List();
14   for(Iterator it = list.iterator(); it.hasNext();)
15       res.add(safeMap(it.next(), transform));
16   return res;
17 }
```

Listing 3.5: Ejemplo 5

Para inferir los requerimientos de memoria de `test` es necesario analizar primero cada una de sus llamadas. Esto nos impone como orden de análisis `copy`, `map`, `safeMap` y finalmente `test`.

El método `copy` sólo crea una copia de la lista de entrada generando una partición residual. Utiliza también un iterador como objeto temporal. Aplicando las reglas mencionadas el resumen de `copy` es `copy`.$Tmp = 1$ y `copy`.$Rsd(ret) = list.size + 1$. El consumo del método `map` fue explicado anteriormente.

Habiendo calculado el resumen para estos dos métodos, se analiza el comportamiento de `safeMap`:

- La partición residual de `copy` no escapa a `safeMap`. Por lo tanto pasa a ser parte de su partición temporal incrementándola en $list.size + 1$.

- La partición residual de `map` es la lista de retorno de `safeMap` por lo que genera un residual de $5 * list.size + 1$. Notar que se está asumiendo el peor caso que fue calculado aplicando las reglas de **acumulación** y **maximización**.

- `map` necesita 3 objetos temporales mientras que `copy` necesita de 1, por lo que la parte temporal de `safeMap` toma el valor de 3 que es el máximo entre ambos.

- Componiendo lo mencionado, `safeMap` puede ser resumido como `safeMap`.$Tmp = list.size + 4$ y `safeMap`.$Rsd(ret) = 5 * list.size + 1$.

Luego, se continúa con el análisis de `test`:

- `test` crea una lista que es retornada incrementando la partición residual en 1.
- El iterador se convierte en temporal incrementando la partición temporal en 1.
- Cada iteración agrega un nuevo nodo en la lista de retorno, por lo que aplicando **acumulación** se obtiene un incremento de $list.size$ en la parte residual.
- Cada iteración necesita de $list.size+4$ objetos a causa del temporal de `safeMap`. Dado que esta expresión temporal depende del tamaño de la lista ingresada como parámetro, se tiene que considerar el peor caso. Esto puede ser realizado maximizando (**maximización**) la parte temporal. Para poder realizarlo se necesita proveer una cota para el máximo tamaño de lista, donde $list(i)$ se refiere a la lista de la i-esima posición de $list$ y $maxSize$ es un parámetro relevante relacionado con el máximo tamaño alcanzable por ellas, el cual puede ser derivado de los parámetros formales aunque no de manera trivial.

$$Max_{i=1}^{list.size}\{list(i).size + 4 \mid 0 \leq list(i).size \leq maxSize\} = maxSize + 4$$

- Cada iteración genera una nueva lista llamando a `safeMap` que se agrega a la lista de retorno, incrementando la parte residual de `test`. Dado que `safeMap`.$Rsd(ret) = 5 * list.size + 1$, aplicando la regla de **acumulación** sobre el espacio de iteración se obtiene la cantidad total de objetos residuales. Luego, la acumulación sobre el espacio de iteración puede ser descripta de la siguiente forma:

$$\sum_{i=1}^{list.size} (5 * list(i).size + 1)$$

Sin embargo, el tamaño de cada sublista no es conocido en tiempo de compilación. Supongamos que existe una función $f(i)$ que describe el tamaño de `list(i)` en la iteración `i`, luego se obtiene la siguiente acumulación simbólica:

$$\sum_{i=1}^{list.size} (5 * f(i) + 1)$$

Si $f(i)$ es conocido en tiempo de compilación entonces la suma simbólica puede ser resuelta. Para hacer una idea de esto, supongamos que $f(i) = i$, luego la acumulación simbólica va a ser la siguiente:

$$\sum_{i=1}^{list.size} (5 * i + 1) = 5 * list.size * (list.size + 1) * 1/2 + list.size$$

Si $f(i)$ es un polinomio y el espacio de iteración puede ser descripto con un conjunto de constantes lineales, entonces la acumulación o sumarización puede ser resuelta [Ver10]. Asumamos que esa función es desconocida, entonces es posible sobreaproximar considerando una cota como se hizo en el paso anterior, obteniendo la siguiente expresión:

$$\sum_{i=1}^{list.size} (5{*}list(i).size{+}1) \leq \sum_{i=1}^{list.size} (5{*}maxSize{+}1) \leq (5{*}maxSize{+}1){*}list.size$$

- Componiendo las explicaciones, `test` puede ser resumido como $\texttt{test}.Tmp = maxSize + 5$ y $\texttt{test}.Rsd(ret) = ((5 * maxSize + 1) * list.size) + list.size + 1$

3.2. Conclusiones

El Ejemplo 5 muestra cómo los requerimientos de memoria dinámica son calculados analizando cada método de manera composicional. Para hacer eso nos basamos en:

- Una técnica para acotar el número de visitas a una instrucción.
- Un oráculo que analice e infiera el tiempo de vida de los objetos.
- Un conjunto de reglas composicionales definidas sobre operaciones simbólicas como maximización, acumulación y supremo de expresiones simbólicas.
- Expresar el consumo en términos de parámetros relevantes derivados de los parámetros formales del método m. Esto permite introducir parámetros relevantes cuya derivación no es trivial pero que influyen en el consumo de m, por ejemplo el $maxSize$.

En el siguiente capítulo se presenta formalmente el análisis composicional y se definen los conceptos que se presentaron informalmente en la presente sección.

Capítulo 4

Análisis composicional del consumo de memoria

En el capítulo anterior presentamos cómo los requerimientos de memoria dinámica son calculados analizando cada método de manera composicional. Para eso nos basamos en una técnica para acotar el número de visitas a una instrucción, un oráculo para analizar e inferir el tiempo de vida de los objetos y un conjunto de reglas composicionales definidas sobre operaciones simbólicas como maximización, acumulación y supremo de expresiones simbólicas.

En este capítulo se presenta un algoritmo composicional para construir resúmenes de consumo de memoria formalizando los conceptos que se presentaron en el capítulo anterior. Es un algoritmo genérico dentro de las limitaciones del modelo de memoria elegido, en el sentido que se basa en otros análisis que puedan ser instanciados para generar diferentes soluciones.

4.1. Infiriendo resúmenes

En esta sección se presenta el algoritmo composicional de inferencia desarrollado a lo largo de este trabajo. Se asume que se cuenta con la implementación de las siguientes herramientas:

- (CT): Es capaz de contar la cantidad de visitas a una instrucción de un programa.
- (SA): Es capaz de operar con expresiones paramétricas.
- (EA): Es capaz de aproximar el tiempo de vida de los objetos.

En la Sección 4.2 se describen las operaciones que deben poseer las herramientas. Es importante notar que las herramientas se asumen implementadas como módulos con estado interno. Esto permite a las mismas haber pre-procesado el programa

```
1:  procedure BuildResourceSummary(m, x̄_m, γ̄_m): S_m
2:      S_m.Tmp := 0 ; S_m.Rsd := ∅
3:      S_m.γ̄ := γ̄_m; S_m.x̄ := x̄_m; S_m.p̄ := params(m)
4:                                                   ▷ Sección intra-procedural
5:      for each as : new T ∈ body(m) do
6:          bound = CT.count(m, params(m), γ̄_m, as)
7:          partition = EA.getPartition(as)
8:          if partition is not Temporal then
9:              S_m.Rsd(partition) := SA.add(S_m.Rsd(partition), bound)
10:         else
11:             S_m.Tmp = SA.add(S_m.Tmp, bound)
12:                                                  ▷ Sección inter-procedural
13:     tempCalls := 0
14:     for each cs : call m'(args) ∈ body(m) do
15:         ⟨tempCall, resCall, resCaptured⟩ = InterProcAnalysis(m, args, γ̄_m, cs)
16:         ▷ Considerar sólo el máximo consumo temporal entre todas las llamadas
17:         tempCalls := SA.join(tempCalls, tempCall)
18:         S_m.Tmp := SA.add(S_m.Tmp, resCaptured)
19:         for each partition ∈ Dom(resCall) do
20:             S_m.Rsd(partition) := SA.add(S_m.Rsd(partition), resCall(partition))

21:     S_m.Tmp := SA.add(S_m.Tmp, tempCalls)
```

Figura 4.1: Algoritmo para construir resúmenes de métodos

previo a la ejecución del análisis de consumo. Por ejemplo, el EA almacena la información del análisis de escape calculado. En la Sección 6.3 se presenta una posible implementación de estas herramientas y, por lo tanto, del algoritmo.

Dado un método m el algoritmo muestra cómo construir un resumen de consumo basándose en:

- La información local de consumo de memoria del método m.
- La información de consumo de memoria de los métodos invocados por m (conociendo los resúmenes para los mismos).

La Figura 4.1 muestra el algoritmo para construir resúmenes y la Figura 4.2 el análisis inter-procedural para analizar las invocaciones a métodos.

```
1: procedure InterprocAnalysis(m, args̄, γ̄m, cs): ⟨tempCallRet, resCallRet, resCapturedRet⟩
2:     resCall := resCallRet := ∅ ; tempCall := resCaptured := 0
3:     callees :=CG.getCallees(cs)                    ▷ Resolución llamados virtuales
4:     for each C.m' ∈ callees do
5:         res_caller,m' := ∅ ; capturedFrom_C.m' := 0
6:             ▷ Considerar máximo consumo temporal entre las implementaciones
7:         tempCall := SA.join(tempCall, S_C.m'.Tmp)
8:                                   ▷ Considerar efecto sobre particiones del caller
9:         for each part_C.m' ∈ Dom(S_C.m'.Rsd) do
10:            hp_caller := EA.bindPartition(part_C.m', args̄, params(C.m'), cs)
11:            if hp_caller is not Temporal then
12:                res_caller,m'(hp_caller) := SA.add(res_caller,m'(hp_caller), S_C.m'.Rsd(part_C.m'))

13:            else
14:                capturedFrom_C.m' := SA.add(capturedFrom_C.m', S_C.m'.Rsd(part_C.m'))

15:        ▷ Tomar el máximo efecto sobre temporal y residual en el caller entre las implementaciones
16:        for each hp_caller ∈ Dom(res_caller,m') do
17:            resCall(hp_caller) := SA.join(resCall(hp_caller), res_caller,m'(hp_caller))
18:        resCaptured := SA.join(resCaptured, capturedFrom_C.m')
19:        ▷ Componer el resultado considerando la cantidad de llamadas a m en cs
20:    tempCallRet := SA.maximize(tempCall, S_C.m'.γ̄, args̄, params(m), γ̄m, cs)

21:    for each hp_caller ∈ Dom(resCall) do
22:        resCallRet(hp_caller) := SA.summate(resCall(hp_caller), S_C.m'.γ̄, args̄, params(m), γ̄m, cs)
23:    resCapturedRet := SA.summate(resCaptured, S_C.m'.γ̄, args̄, params(m), γ̄m, cs)
```

Figura 4.2: Binding inter-procedural de resúmenes de métodos

El algoritmo de la Figura 4.1 se encuentra dividido en dos secciones principales:

1. El análisis local o intra-procedural.
2. El análisis de las invocaciones a métodos o inter-procedural.

El análisis intra-procedural infiere el consumo de memoria para el método. Esto se realiza básicamente contando visitas a instrucciones que crean nuevos (`new`) objetos (línea 6) y determinando a qué partición pertenecen los objetos creados por dichas instrucciones (línea 7) para sumar el resultado en donde corresponda (líneas 9 y 11). En caso de tratarse de una partición residual se acumulará el resultado en la memoria residual asociada a esa partición (línea 9), mientras que de ser temporal se acumulará en Temporal (línea 11). El conteo se realiza con la ayuda de una herramienta CT que, recordemos, es capaz de contar la cantidad de visitas a una instrucción. Para determinar la partición a la que contribuyen los objetos creados por las instrucciones `new`, el algoritmo se basa en la herramienta EA, un *oráculo* capaz de analizar el tiempo de vida de los objetos. Para acumular el resultado de una partición se utiliza el SA, herramienta capaz de operar con expresiones paramétricas.

La parte inter-procedural es usada para analizar el consumo de cada método invocado (línea 15). Esta información se obtiene a partir los resúmenes de los mismos. El procedimiento `InterproceduralAnalysis` en la Figura 4.2 es el responsable de procesar la invocación a un método.

Este análisis tiene en cuenta la posibilidad de que ocurra una invocación virtual, es decir, que no sea posible determinar de forma estática el método que realmente será invocado dada una instrucción `call`. Es por esto que analiza de forma conservadora el conjunto de los posibles candidatos (línea 3). Con este objetivo, el primer paso es obtener el supremo entre todas las particiones temporales de los candidatos (línea 7).

Luego cada partición de un candidato se vincula, utilizando $EA.bindPartition$, a una partición del método a resumir (línea 10). De esta forma se puede determinar para cada candidato si los objetos representados por las particiones de su resumen serán capturados por el método a resumir y aportan al temporal (línea 14), o bien escapan y aportan al residual de la partición con la que se vinculan (línea 12).

Es importante entender que el algoritmo es conservador en los casos donde se produzca una invocación virtual. Para esto, el consumo temporal producto de capturar objetos que escapan al método invocado se calcula por cada candidato (línea 14) y luego se obtiene el supremo de estas expresiones (línea 18). En el caso del consumo residual, el algoritmo calcula el aporte de cada candidato a las distintas particiones del método resumido (línea 12), quedándose con el supremo para cada una de ellas (línea 17).

Una vez procesados los distintos resúmenes por implementación se dispone de la información necesaria para calcular el resultado del análisis inter-procedural. Los pasos finales consisten en dar el valor del temporal y residual que produce la invocación sobre el método a resumir. El tratamiento del temporal es diferente del residual debido a que los objetos que conforman el primero pueden ser liberados al finalizar la ejecución del método invocado. Consideremos un método m' que es invocado k

veces. Cada vez que m' es ejecutado creará a lo sumo $S_{m'}.Tmp$ objetos temporales, liberándolos cuando m' finalice su ejecución. Esto significa que un método llamador debe reservar solamente espacio para la llamada más costosa dentro de las k realizadas (Ver Capítulo 3). Este cálculo es realizado por $SA.maximize$ que obtiene una expresión paramétrica, *en término de los parámetros relevantes del llamador*, aproximando el consumo máximo entre todas las llamadas a m' en el punto de programa cs (línea 20).

Por el contrario, las particiones residuales tienen un efecto acumulativo. Como consecuencia, el llamador debe considerar el consumo residual de cada uno de los llamados. Esto es realizado por la operación $SA.Summate$ que obtiene una expresión paramétrica, en término de los parámetros relevantes del llamador, aproximando el consumo acumulado de *todas las llamadas* a m' en el punto de programa cs (líneas 23 y 22).

Finalmente, a partir de la línea 17 del algoritmo de la Figura 4.1 se ve como se utiliza la información calculada por el análisis inter-procedural de una llamada. Por un lado, se calcula el supremo entre el temporal de las llamadas anteriores y el de la llamada actual (línea 17). Alcanza con considerar el supremo de las expresiones ya que los objetos asignados a la partición temporal del método invocado son liberados al finalizar la invocación y por lo tanto nunca compartirán el mismo tiempo de vida. Luego se acumula el residual capturado (línea 18) que no es otra cosa que el consumo producto de capturar objetos que exceden el tiempo de vida de los métodos invocados. Como cada invocación produce un nuevo conjunto de objetos residuales, es necesario acumular el valor de dichas expresiones a lo largo de cada invocación. Por último se acumula el residual de la llamada realizada, que también resulta residual para el llamador, ya que excede su tiempo de vida (línea 20).

Notar que una partición proveniente de un método invocado no puede ser repartida en varias particiones del método llamador. Esto es consecuencia de que una partición sea la mínima unidad de información de tiempo de vida. Es aquí donde puede apreciarse cómo la cantidad de particiones a considerar define la granularidad del análisis y del resumen.

Luego, para un método m se logra inferir un resumen formado por:

- El residual del método m, dividido en las particiones correspondientes, producto de la acumulación de los objetos creados por m que exceden su tiempo de vida y de los objetos creados por las invocaciones de m que exceden el tiempo de vida de m (líneas 9 y 20).
- La partición temporal del método m compuesta por:
  - Los objetos creados por m que no exceden su tiempo de vida (línea 11).
  - El supremo de los temporales de las invocaciones realizadas por m (línea 17).
  - La acumulación de los residuales de las invocaciones de m que no exceden su tiempo de vida (línea 18).

Ejemplo Veamos el funcionamiento del algoritmo utilizando el método `map` y las dos clases transformadoras presentadas en el Código 4.1.

```
0 List map(List list, Trans transform)
    {
1   List res = new List();
2   for (Iterator it = list.iterator();
        it.hasNext();) {
3     Object o = transform.apply(it.
          next());
4     res.add(o);
5   }
6   return res;
7 }
8 class Trans {
9  Object apply(Object object) {
10   Date date = new Date();
11   Random random = new Random(date.
        getTime());
12   int value = ((Integer) object).
        intValue();
13   return new Integer( value * random
        .nextInt());
14  }
15 }
16 class Trans2 extends Trans {
17  Object apply(Object object) {
18   Date date = new Date();
19   Random random = new Random(date.
        getTime());
20   int value = ((Integer) object).
        intValue()
21   return new Integer[]{
22  new Integer(value),
23  new Integer(value * random.nextInt()
      )};
24  }
25 }
```

Listing 4.1: Ejemplo map y clases Transformadoras.

Supongamos que el algoritmo ya ha calculado los resúmenes para `Trans.apply` y `Trans2.apply` obteniendo los siguientes resultados:

- `Trans.apply`: $S.Tmp = 2$, $S.Rsd(ret_{13}) = 1$.
- `Trans2.apply`: $S.Tmp = 2$, $S.Rsd(ret_{21}) = 2$, $S.Rsd(ret_{22}) = 1$, $S.Rsd(ret_{23}) = 1$.

La granularidad utilizada para ambos métodos es de una partición por instrucción de tipo `new` y se la nota como ret_{linea}.

Además, supongamos que el resumen de `map` posee una única partición residual ret_{map} la cual representa los objetos que escapan a `map`. Notar que el criterio de selección de granularidad utilizado en `map` difiere al utilizado previamente. Si bien esto no es un caso usual, no hay ninguna restricción que impida que ocurra. Además, la definición de $EA.bindPartition$ permite que varias particiones de un candidato sean vinculadas a una misma partición residual del método a resumir. Por esta razón el algoritmo de la Figura 4.2 *agrupa* el aporte de cada partición del candidato en la partición residual correspondiente del método resumido (línea 12). El ejemplo permite comprender un aspecto importante del algoritmo: cómo operar cuando varias particiones de un método invocado son vinculadas por el EA a una misma partición del método llamador.

Al finalizar la sección intra-procedural del algoritmo de la Figura 4.1, es decir, luego de procesar los `new` del método `map`, se obtiene que $S.Tmp = 0$ y $S.Rsd(ret_{map}) = 1$. Esto es producto del `new` de la línea 1. Luego se procede a la sección inter-procedural que procesa las invocaciones a los métodos `List.iterator`, `List.add`, `Trans.apply`, `Iterator.next`, `Iterator.hasNext` y al constructor de la clase `List` realizando llamadas al algoritmo presentado en la Figura 4.2 por cada una de ellos. Analicemos el caso de la invocación a `apply`. La misma es una invocación virtual, por lo tanto el algoritmo itera por todas las posibles implementaciones de este método. En este caso las implementaciones posibles se encuentran en las clases `Trans` y `Trans2`. En la línea 7 se obtendrá que el máximo consumo temporal entre ellas es $tempCall = 2$.

Luego a partir de la línea 10 se itera por todas las particiones del resumen de una implementación. Analicemos el caso de `Trans2.apply` que cuenta con más de una partición. $EA.bindPartition$ nos indica la partición de destino en el método llamador para estas particiones. Esta partición de destino puede o no ser capturada en el método llamador, en este caso ninguna es capturada. Como supusimos, todas las particiones de `Trans2.apply` tienen como destino la misma partición en el método `map`, es decir ret_{map}. Luego todas ellas aportarán al consumo de ret_{map} y su consumo deberá ser acumulado en esta partición. Esto ocurre en la línea 12. Si bien en este caso aportan a la misma partición y ésta escapa, podrían aportar a particiones diferentes (línea 12) o bien no escapar al método llamador y aportar al residual capturado (línea 14).

Al considerar el método `Trans.apply` vemos que la partición ret_{13} también aporta a la partición ret_{map}. Como se explicó anteriormente, sólo debemos considerar la partición de la implementación cuyo aporte sea mayor (línea 17), en este caso el aporte de `Trans2.apply` que será de 4. Además, como ninguna partición de las implementaciones es capturada por el método llamador, el residual capturado será $resCaptured = 0$. Luego en la línea 20 se maximiza el valor de $tempCall$ calculado previamente ya que el mismo es ejecutado varias veces dentro de un ciclo, en par-

ticular como se maximiza un valor constante se obtiene $tempCallRet = 2$. Además en la línea 22 se obtiene que $retCallRet(ret_{map}) = 4 * list.size$, ya que la cantidad de veces que es ejecutada la partición calculada previamente se la relaciona con la expresión $list.size$.

Continuando con el resto de las invocaciones realizadas por el método `map`, se obtiene que el resumen final de este método es:

$$\langle [list, transform], [list.size], [\gamma_{map}], 3, Rsd \rangle$$

donde $\gamma_{map}(list, transform) = list.size$ y $Rsd(ret_{map}) = 5 * list.size$. Recordemos que los parámetros relevantes y γ_{map} se asumen dados.

4.2. Herramientas auxiliares

A continuación se presenta la definición formal de las operaciones con las que debe contar cada una de las herramientas auxiliares asumidas como existentes durante el algoritmo presentado.

4.2.1. EA

El análisis asume la existencia de un mecanismo capaz de calcular las particiones del *heap* a considerar. Cualquier técnica capaz de aproximar el tiempo de vida de los objetos como escape [SR01, SYG05] o análisis de regiones [CR04] puede realizar el trabajo. El algoritmo utiliza un *oráculo* llamado EA que determina el tiempo de vida de un objeto. Según lo especificado en las secciones anteriores, las operaciones requeridas por parte del EA son:

- $getPartition$: $Labels \mapsto \mathsf{HP} \cup \mathsf{Temporal}$, que determina en qué partición es asignado un objeto creado localmente (instrucción `new`).
- $bindPartition$: $\mathsf{HP} \times \bar{Args} \times \bar{P} \times Labels \mapsto \mathsf{HP}$, que dado una partición (e.g. una partición de un método invocado), la lista de argumentos del método y los parámetros del mismo determina la partición de destino (i.e. la partición del llamador) de todos los objetos de la partición dada como parámetro.

Recordemos que una partición proveniente de un método llamado no puede ser partida en varias particiones en el método llamador.

4.2.2. CT

El análisis requiere una técnica capaz de calcular el número de visitas a un punto de programa dado. Esta es la base para calcular el consumo de memoria. Afortunadamente existen diferentes técnicas capaces de calcular una sobreaproximación del

número de visitas [AAG+07, GMC09, GZ10, BGY06]. Llamamos a este módulo CT, el cual provee la siguiente funcionalidad:

- *count* : $Methods \times \bar{P} \times \bar{\Gamma} \times Labels \mapsto \mathbb{E}[\bar{X}]$, que dado un método m, una lista de parámetros formales $\bar{p_m}$, un mapping $\bar{\gamma_m}$ de parámetros formales $\bar{p_m}$ a parámetros relevantes $\bar{x_m}$ y un punto del programa l devuelve una expresión paramétrica en término de parámetros relevantes $\bar{x_m}$, delimitando el número de veces que l es visitado en cada ejecución del método m.

El algoritmo asume que solamente se requiere el método y el punto del programa, pero esto puede ser adaptado a las necesidades de otras técnicas. Por ejemplo [BGY06] puede requerir un invariante que describa las potenciales asignaciones de variables en ese punto. Otros pueden requerir funciones variantes [AAG+07] o contadores especializados [GMC09].

4.2.3. SA

Finalmente, el análisis asume la existencia de una calculadora simbólica capaz de operar con expresiones paramétricas. La calculadora trabaja con expresiones paramétricas que componen un semi-reticulado donde el operador de *join* debe devolver una cota superior de las expresiones sobre las que se aplica, intentando que sea la mínima. Llamamos a esta calculadora SA y debe ser capaz de proveer el siguiente conjunto de operaciones:

- *summate*: $\mathbb{E}[\bar{X}'] \times \bar{\Gamma}' \times A\bar{r}gs \times \bar{P} \times \bar{\Gamma} \times Label \mapsto \mathbb{E}[\bar{X}]$, que requiere para realizar la operación:

  - Una expresión paramétrica e en términos de $\bar{x_{m'}}$.
  - Un mapping $\bar{\gamma_{m'}}$ de parámetros formales $\bar{p_{m'}}$ a parámetros relevantes $\bar{x_{m'}}$ del método invocado en la instrucción asociada al label cs.
  - Una lista $a\bar{r}gs$ de argumentos con los que se realiza la invocación.
  - Una lista $\bar{p_m}$ de parámetros formales del método en el que se produce la invocación.
  - Un mapping $\bar{\gamma_m}$ de parámetros formales $\bar{p_m}$ a parámetros relevantes $\bar{x_m}$ del método en el que se produce la invocación.
  - El label cs en el que se produce la invocación.

  La operación obtiene una nueva expresión paramétrica en términos de $\bar{x_m} = \bar{\gamma_m}(\bar{p_m})$, sobreaproximando la acumulación de la expresión $e(\bar{\gamma_{m'}}(a\bar{r}gs))$ (donde $\bar{x_{m'}} = \bar{\gamma_{m'}}(a\bar{r}gs)$) para todas las visitas al punto cs del programa. Notar que por un lado la expresión e está en términos de los parámetros formales del método invocado, mientras que el resultado de la operación debe devolverse

en términos de parámetros relevantes del método llamador. Este cambio de parámetros requiere calcular la relación entre $\bar{args}$ y $\bar{p_m}$. Como la noción de parámetro y argumento relevante es propia del análisis realizado, para facilitar la construcción de implementaciones de la herramienta que calculen esta relación, se decidió incluir en los argumentos de la operación a los parámetros y argumentos con sus respectivos mappings para que la implementación haga las transformaciones que requiera.

- *maximize*: $\mathbb{E}[\bar{X}'] \times \bar{\Gamma}' \times \bar{Args} \times \bar{P} \times \bar{\Gamma} \times Label \mapsto \mathbb{E}[\bar{X}]$, cuya firma resulta análoga a *summate* y sobreaproxima el máximo valor de la expresión $e(\bar{\gamma_{m'}}(\bar{args}))$ para todas las visitas al punto cs del programa.
- *join*: $\mathbb{E}[\bar{X}] \times \mathbb{E}[\bar{X}] \mapsto \mathbb{E}[\bar{X}]$, que dado dos expresiones paramétricas e_1 y e_2 obtiene una expresión mayor o igual a ambas expresiones.
- *add*: $\mathbb{E}[\bar{X}] \times \mathbb{E}[\bar{X}] \mapsto \mathbb{E}[\bar{X}]$, que dado dos expresiones paramétricas e_1 y e_2 obtiene la suma de ambas expresiones.

4.3. Considerando invarianza de ciclo

En la sección anterior se definió el comportamiento de la operación *join* del SA. Como se verá en la Sección 6.3.2, cada vez que la operación de *join* es aplicada, la calculadora utilizada puede generar una sobreaproximación del resultado esperado. Esto se debe a que trabaja con polinomios y, como se menciona durante el Capítulo 1, la maximización de polinomios es un problema complejo que se aborda dando una cota para el máximo (ver Sección 1.2).

Resulta interesante minimizar el impacto que produce en las expresiones la aplicación del *join*, siendo que es la operación más utilizada por el algoritmo y la que con menor precisión se resuelve (ver Sección 6.3.2). En esta sección se presenta una versión optimizada del algoritmo en caso que se sepa que la invocación se hace siempre sobre el mismo objeto (i.e. el objeto sobre el cual se realiza la invocación es *loop-invariant*).

Durante la presentación del algoritmo se habló de invocaciones virtuales y la imposibilidad de conocer de forma estática el método que efectivamente será invocado. Si la invocación ocurre dentro de un ciclo, el algoritmo asume que en cada invocación dentro del ciclo puede ser ejecutada una implementación distinta. Esto obliga a utilizar el *maximize* y *summate* después del *join*. En casos donde sea posible asegurar que la invocación es *loop-invariant*, si bien todavía se desconoce la implementación que será invocada, se puede asumir que siempre será la misma. Esto permite efectuar las operaciones de *maximize* y *summate* antes de realizar el *join*. Esto ocurre porque en cada iteración del ciclo la invocación será realizada sobre el mismo obje-

1: **procedure** InterprocAnalysis(m, $\bar{args}$, $\bar{\gamma_m}$, cs): $\langle tempCallRet, resCallRet, resCapturedRet\rangle$

2: $resCall := resCallRet := \emptyset$; $tempCall := resCaptured := 0$

3: $callees$:=CG.getCallees(cs)

4: **for** each $C.m' \in callees$ **do**

5: $res_{caller,m'} := \emptyset$; $capturedFrom_{C.m'} := 0$

6: $tempCall := SA.join(tempCall, SA.maximize(\mathtt{S}_{C.m'}.Tmp, \mathtt{S}_{C.m'}.\bar{\gamma}, \bar{args}, params(m), \bar{\gamma_m}, cs))$

7: **for** each $part_{C.m'} \in Dom(\mathtt{S}_{C.m'}.Rsd)$ **do**

8: $hp_{caller} := EA.bindPartition(part_{C.m'}, \bar{args}, params(C.m'), cs)$

9: **if** hp_{caller} is not Temporal **then**

10: $res_{caller,m'}(hp_{caller}) := SA.add(res_{caller,m'}(hp_{caller}), SA.summate(\mathtt{S}_{C.m'}.Rsd(part_{C.m'}), \mathtt{S}_{C.m'}.\bar{\gamma}, \bar{args}, params(m), \bar{\gamma_m}, cs))$

11: **else**

12: $capturedFrom_{C.m'} := SA.add(capturedFrom_{C.m'}, SA.summate(\mathtt{S}_{C.m'}.Rsd(part_{C.m'}), \mathtt{S}_{C.m'}.\bar{\gamma}, \bar{args}, params(m), \bar{\gamma_m}, cs))$

13:

14: **for** each $hp_{caller} \in Dom(res_{caller,m'})$ **do**

15: $resCall(hp_{caller}) := SA.join(resCall(hp_{caller}), res_{caller,m'}(hp_{caller}))$

16: $resCaptured := SA.join(resCaptured, capturedFrom_{C.m'})$

17:

18: $tempCallRet := tempCall$; $resCapturedRet := resCaptured$

19: **for** each $hp_{caller} \in Dom(resCall)$ **do**

20: $resCallRet(hp_{caller}) := resCall(hp_{caller})$

Figura 4.3: Binding inter-procedural de resúmenes de métodos considerando invarianza de ciclo

to. Por tanto alcanza con calcular el temporal (usando el *maximize*) y el residual (usando el *summate*) para luego quedarnos con el peor escenario (usando el *join*).

La versión optimizada del algoritmo se presenta en la Figura 4.3. Como se puede ver las operaciones de *summate* y *maximize* se hacen antes de tomar el supremo por implementación (líneas 6, 10 y 12).

La presencia de *loop-invariant* puede especificarse manualmente o bien utilizando cualquier análisis existente para detectar el mismo. En caso de querer utilizarse debe introducirse un oráculo capaz de reconocer si una invocación se lleva a cabo en un punto del programa donde existe la propiedad de *loop-invariant*.

En caso que el operador *join* pudiese calcular el supremo de forma exacta invertir

el orden de los operadores no generaría ningún beneficio, sin embargo cuando la implementación del operador $join$ es imprecisa, invertir el orden en que son aplicadas las operaciones (i.e. pasar de $summate(join)$ a $join(summate)$) minimiza el impacto producido por la imprecisión del mismo.

4.4. Conclusiones

En este capítulo presentamos un algoritmo composicional para construir resúmenes de consumo de memoria y definimos los conceptos presentados informalmente en el capítulo anterior. Para construir los resúmenes definimos una serie de procedimientos, uno de ellos capaz de inferir el consumo de memoria para un método (4.1), y otro capaz de analizar el consumo de memoria de cada método invocado (4.2).

Asumimos la existencia de una serie de herramientas auxiliares (CT, SA y EA) y definimos las operaciones que las mismas deben proveer. Además presentamos una forma de optimizar el algoritmo composicional en los casos que estemos en presencia de *loop-invariant*.

En el capítulo siguiente se presenta el análisis de escape basado en [SR05b, SR05a] utilizado como parte de la instanciación realizada.

Capítulo 5

Análisis de escape

En el capítulo anterior presentamos un algoritmo composicional para construir resúmenes de consumo de memoria. Para ello definimos una serie de procedimientos capaces de inferir el consumo de memoria para un método y de analizar el consumo de memoria de cada método invocado. Asumimos la existencia de una serie de herramientas auxiliares (CT, SA y EA) y definimos las operaciones que las mismas deben proveer.

En este capítulo se presenta el análisis de escape basado en [SR05b, SR05a], utilizado como parte de la instanciación realizada y una adaptación del mismo para ser k-sensitivo. Como el análisis de [SR05b, SR05a] no es una contribución de este trabajo, la lectura de la Secciones 5.1 a 5.4 no es indispensable. La Sección 5.5 presenta la adaptación del análisis realizada para dar el efecto de k-sensitividad.

La técnica presenta un análisis composicional el cual permite obtener para cada método m un resumen que representa el heap accedido por dicho método. Además permite calcular el conjunto de campos externos al método que son modificados (característica no utilizada en este trabajo).

El resumen, en principio, puede ser entendido como un grafo dirigido donde los nodos representan objetos del heap y los ejes referencias entre los mismos, razón por la cual estarán etiquetados con el campo a través del cual se relacionan. Tanto los nodos como los ejes son tipados. Los nodos pueden representar parámetros (*parameter nodes*), objetos creados por el método analizado (*inside nodes*) u objetos leídos que viven más alla del mismo (*load nodes*). A su vez, los ejes pueden representar referencias generadas por el método bajo análisis (*inside edges*) o bien referencias a objetos que escapan al mismo (*outside edges*). Se utiliza $\langle n_1, f, n_2 \rangle$ para modelar un eje de n_1 a n_2 etiquetado con el campo f.

```
0 class List {
1  Cell head = null
2  void add(Object e) {
3   head = new Cell(e, head);
4  }
5  Iterator iterator() {
6     return new ListItr(head);
7   }
8 }
9
10 class Cell {
11  Object data;
12  Cell  next;
13  Cell(Object d, Cell n) {
14   data = d; next = n;
15  }
16 }
17
18 interface Iterator{
19  boolean hasNext();
20  Object next();
21 }
22
23 class ListItr implements Iterator {
24  Cell cell;
25  ListItr(Cell head){
26   cell = head;
27  }
28
29  public boolean hasNext(){
30   return cell != next;
31  }
32
33  public Object next(){
34   Object result = cell.data;
35   cell = cell.next;
36   return result;
37  }
38 }
39
40 class Trans {
41  Object apply(Object object) {
42    Date date = new Date();
43    Random random = new Random(date.
        getTime());
44    int value = ((Integer) object).
        intValue();
45    return new Integer( value * random
        .nextInt());
46 }
47
48 List map(List list, Trans transform)
    {
49   List res =  new List();
50   for (Iterator it = list.iterator();
        it.hasNext();) {
51     Object o = transform.apply(it.
        next());
52     res.add(o);
53   }
54   return res;
55 }
56
57 List copy(List list) {
58   List res = new List();
59   for (Iterator it = list.iterator();
        it.hasNext();)
60     res.add(it.next());
61   return res;
62 }
63
64 List safeMap(List list,Trans
      transform) {
65   List cp = copy(list);
66   return map(cp, transform);
67 }
68
69 List test(List<List> list, Trans
      transform){
70   List res = new List();
71   for(Iterator it = list.iterator();
        it.hasNext();)
72     res.add(safeMap(it.next(),
          transform));
73   return res;
74 }
```

Listing 5.1: Código de ejemplo para el Capítulo 5.

El análisis considera que un objeto escapa si puede ser accedido fuera del contexto del método que lo generó. A nivel resumen, un *inside node* escapa si existe un camino hasta el, por ejemplo, desde algún parámetro.

El hecho de representar las relaciones con ejes dirigidos otorga al análisis mayor precisión a la hora de modelar las relaciones entre objetos en comparación a otras técnicas que utilizan grafos no dirigidos. Esta característica es la que permite modelar de forma correcta problemas clásicos como la iteración de colecciones. Los iteradores

son objetos locales que se relacionan con otros objetos que rara vez conocen la existencia del iterador. Por lo tanto, es razonable creer que el nodo que represente a un iterador tendrá sólo ejes que salen de él. De esta forma se observa que no hay caminos que puedan llegar al nodo que representa al iterador, razón por la cual será considerado como un objeto temporal. Esto no ocurre si se usa utiliza un grafo no dirigido y es la razón por la cual las técnicas basadas en ellos no modelan bien el patrón descripto.

Al igual que el análisis de consumo los métodos se analizan bottom-up, comenzando por las hojas del call graph. El resumen es generado sin conocer el contexto en que el método m es invocado. Cuando se analiza el efecto de una invocación a m, el resumen es instanciado utilizando la información que brindan los distintos tipos de nodos y ejes para resolver problemas como el aliasing. A diferencia del análisis de consumo, el análisis de escape soporta recursión.

En las siguientes subsecciones se da una explicación más detallada de los aspectos relevantes a los efectos del algoritmo de consumo, así como también se muestra un ejemplo para entender los conceptos claves. Para ello se utilizan parte de los ejemplos presentados en el Capítulo 3 junto con la definición de las clases `List`, `Cell` y `ListItr` presentadas en la Figura 5.1. La clase `List` implementa una lista enlazada usando instancias de la clase `Cell`. Soporta las operaciones `add(e)`, que agrega el objeto `e` a la lista, y `iterator()`, que retorna un iterador sobre los elementos de la lista.

5.1. Definiciones

A continuación se presentan algunas definiciones que permiten comprender la técnica de escape presentadas en [SR05b, SR05a].

Definición: Points-To Graph Un Points-To Graph (PTG), en el contexto de este trabajo, es un grafo dirigido en donde un nodo modela objetos del programa analizado y un eje modela referencias al *heap*. Se utiliza $\langle n_1, f, n_2 \rangle$ para notar un eje de n_1 hacia n_2 etiquetado con el campo f. Intuitivamente este eje modela una referencia de un objeto al que n_1 modela hacia un objeto al que n_2 modela, a través de un campo f.

Se representa un Points-To Graph con la tupla $\langle I, O, L, E \rangle$. I es el conjunto de los *inside edges*, donde un *inside edge* representa una referencia al heap creada en el contexto de análisis. O es el conjunto de los *outside edges*, donde un *outside edge* representa una referencia al heap creada por la lectura de un campo de un objeto que escapa al contexto de análisis, y por su naturaleza, siempre apunta a un *load node*. L es un mapping que representa el estado de las variables locales del método, donde $L(v)$ es el conjunto de nodos a los que la variable local v puede apuntar (ver Sección 5.3), al que se lo puede pensar como el punto de entrada al grafo. $L(v_{ret})$ es el conjunto

de nodos que un método m podría retornar. E representa al conjunto de nodos que escapan globalmente, es decir, aquellos nodos que son pasados como parámetro a un método no analizable o bien cuya referencia es guardada en una variable estática.

$INode$, $PNode$ y $LNode$ representan a los conjuntos de *inside node*, *parameter node* y *load node* respectivamente. Un *inside node* $n^I_{lb} \in INode$ representa a todos los objetos creados por la ejecución de la instrucción identificada por el label lb, es decir, se introduce un *inside node* por cada label que corresponda a una instrucción de tipo `new`. Un *parameter node* $n^P_{m,i} \in PNode$ representa un parámetro formal p_i del método m, es decir, dada una invocación a un método representa un único objeto, el apuntado por el argumento pasado como parámetro a la invocación. Un *load node* $n^L_{lb} \in LNode$ representa a todos los objetos leídos por la ejecución de la instrucción identificada por el label lb desde un objeto que escapa al método bajo análisis. Existen instrucciones de tipo LOAD (i.e. $a = b.f$ o $a = b[i]$) que pueden leer referencias de objetos que escapan al método bajo análisis, es decir, de objetos accesibles por fuera del contexto de análisis. Es por esto que se introduce un *load node*, ya que el mismo representa una marca que permite informar al llamador del método que se ha hecho una lectura de un campo cuyo contenido es desconocido. Existe un nodo especial denominado *global node* n_{GBL} el cual representa a los objetos que pueden ser accedidos desde cualquier parte del programa, es decir, se utiliza para modelar objetos leídos de campos estáticos o bien objetos retornados por métodos no analizables. Se utiliza $Node$ para notar al conjunto de todos los nodos del Points-To Graph, es decir $Node = INode \cup PNode \cup LNode \cup \{n_{GBL}\}$

Definición: $\sqsubseteq$ de L. Sean L_1 y L_2 mappings que representan el estado de las variables locales de un método m,

$$L_1 \sqsubseteq L_2 \ sii \ \forall \ v \in Var \ L_1(v) \subseteq L_2(v)$$

Definición: *join* de L. Sean L_1 y L_2 mappings que representan el estado de las variables locales de un método m,

$$L_1 \ join \ L_2 = \lambda v.(L_1(v) \cup L_2(v))$$

Definición: $\perp$ de PTG (mínimo del reticulado)

$$\perp_{PTGraph} = \langle \emptyset, \emptyset, \lambda v.\emptyset, \emptyset \rangle$$

Definición: El Points-To graph forma un reticulado con $\sqsubseteq$ y *join*.

Definición: $\sqsubseteq$ de PTG

$$\langle I_1, O_1, L_1, E_1 \rangle \sqsubseteq \langle I_2, O_2, L_2, E_2 \rangle \ sii \ \langle I_1 \subseteq I_2, O_1 \subseteq O_2, L_1 \sqsubseteq L_2, E_1 \subseteq E_2 \rangle$$

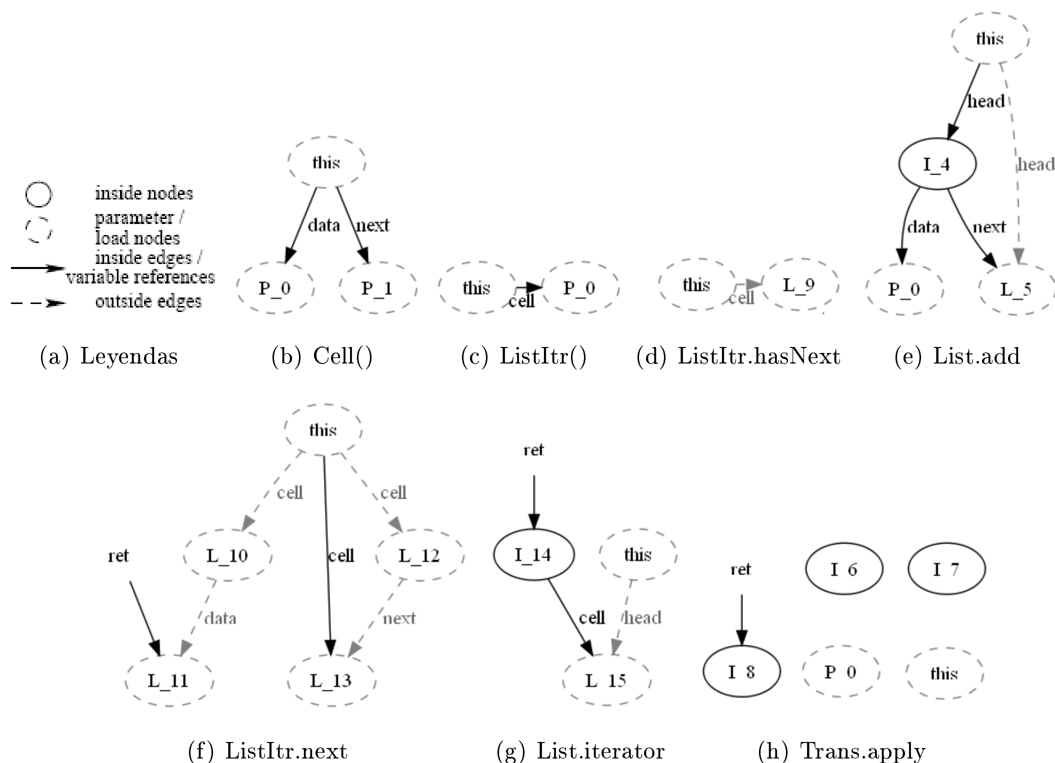
Definición: *join* **de** *PTG*

$$\langle I_1, O_1, L_1, E_1 \rangle join \langle I_2, O_2, L_2, E_2 \rangle = \langle I_1 \cup I_2, O_1 \cup O_2, L_1 \ join \ L_2, E_1 \cup E_2 \rangle$$

Definición: Escape Dado un Points-To graph $G = \langle I, O, L, E \rangle$ y un nodo $n \in Node$, $escape(G)(n)$ será verdadero si y solo si n es alcanzable a partir de un nodo perteneciente a $PNode \cup L(v_{ret}) \cup E \cup \{n_{GBL}\}$ a través de un camino de ejes, posiblemente vacio, pertenecientes a $I \cup O$. Si $escape(G)(n)$ es verdadero diremos que n escapa a G, caso contrario diremos que es capturado. □

5.2. Análisis del ejemplo

La Figura 5.1 presenta los resultados del análisis de los métodos llamados por `map` en el Código 5.1. Por simplicidad la única variable representada es la correspondiente al objeto retornado `ret`. La Figura 5.1(b) presenta el análisis para el constructor de la clase `Cell`. El análisis usa los *parameter nodes* de nombre *this*, $P0$ y $P1$ para representar a los objetos que apuntan los parámetros *this*, d y n respectivamente. El análisis usa *inside edges* para modelar las referencias que el constructor crea de *this* a $P0$ y $P1$.



(a) Leyendas (b) Cell() (c) ListItr() (d) ListItr.hasNext (e) List.add

(f) ListItr.next (g) List.iterator (h) Trans.apply

Figura 5.1: Resultados análisis para llamadas del método `map`

La Figura 5.1(e) presenta el análisis para el método `List.add(Object e)`. El método lee el campo `head` del parámetro `this`, y como el análisis no conoce donde

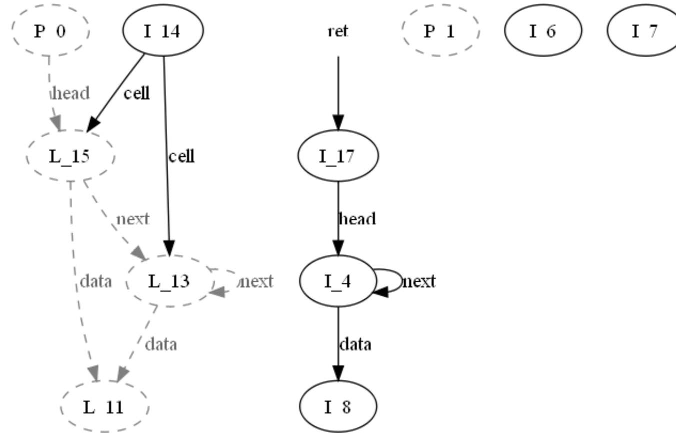
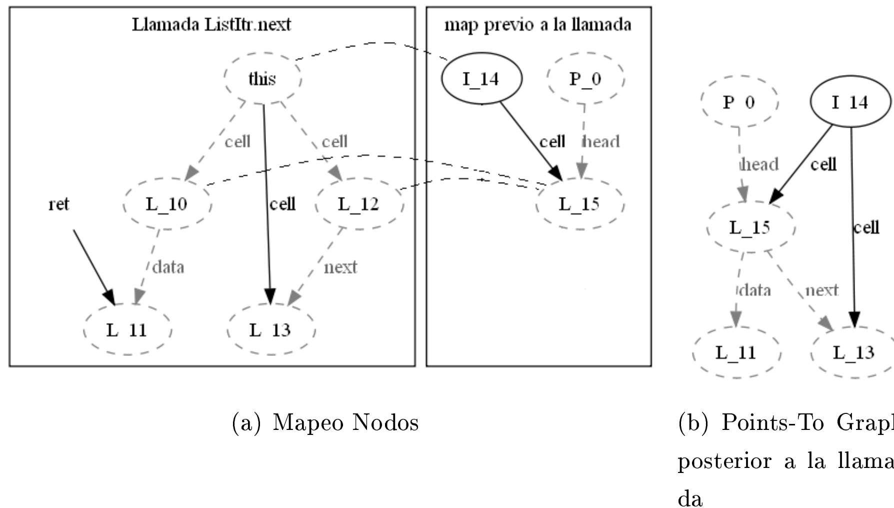


Figura 5.2: Resultados análisis para el método `map`

apunta `this.head` en el contexto de llamada, utiliza el *load node* $L5$ para modelar el objeto cargado y agrega el *outside edge* $\langle this, head, L5\rangle$. Luego el método realiza `new Cell` que se modela con el *inside node* $I4$, e invoca al constructor de `Cell` con los parámetros $I4$, $P0$ y $L5$. Utilizando el Points-To graph previo a la llamada y el del constructor de `Cell` (Figura 5.1(b)), el análisis mapea cada *parameter node* del constructor con uno o más nodos correspondientes del contexto de llamada. En este caso se mapea *this* con $I4$, $P0$ con $P0$, y $P1$ con $L5$. Además se utiliza el mapeo de nodos para incorporar información del Points-To graph del constructor de `Cell` ya que el *inside edge* $\langle this, data, P0\rangle$ se traduce en el *inside edge* $\langle I4, data, P0\rangle$, y de forma similar se agrega $\langle I4, next, L5\rangle$. Finalmente se agrega un *inside edge* $\langle this, head, I4\rangle$.

La Figura 5.1(h) presenta el análisis para el método `Trans.apply(Object object)`. Se pueden ver nodos que no contienen ejes correspondientes al parámetro *this* y *object* y a la creación de los objetos *date* ($I6$) y *random* ($I7$). El hecho que los *inside nodes* no se encuentren conectados con parámetros o con objetos retornados nos da la pauta que son objetos temporales que sólo viven dentro del método. Por último la variable `ret` apunta al objeto creado retornado. De manera similar se analizan los métodos de las Figuras 5.1(c), 5.1(d), 5.1(f), 5.1(g)

El análisis del método `map(List, Trans)` (Figura 5.2) comienza con el parámetro formal `list` apuntando al nodo correspondiente $P0$. Luego crea una lista a retornar correspondiente al nodo $I17$, y llama a `list.iterator()` para obtener un iterador sobre la lista recibida como parámetro. Toma los resultados del método `iterator()` (Figura 5.1(g)), mapea *this* con $P0$ y produce el Points-To graph previo a la llamada a `ListItr.next` que se puede ver en la parte derecha de la Figura 5.3(a). El nodo $I14$ corresponde al iterador retornado por `iterator()`. Luego, el análisis itera sobre el ciclo de las líneas 50-53 hasta obtener un punto fijo.



(a) Mapeo Nodos

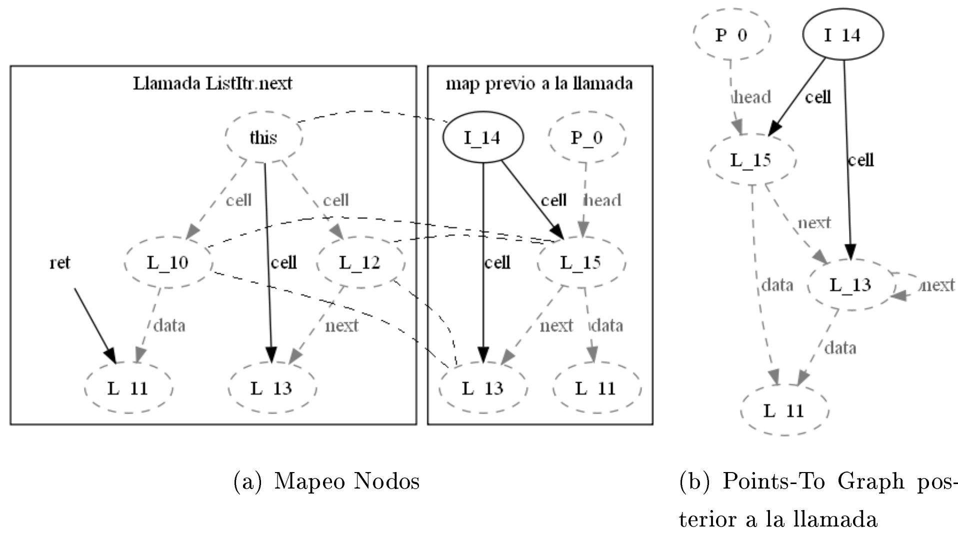
(b) Points-To Graph posterior a la llamada

Figura 5.3: Análisis inter-procedural para el llamado a `ListItr.next` de la primera iteración del ciclo

La Figura 5.3 presenta el análisis inter-procedural para la llamada a `next()` en la primera iteración sobre el cuerpo del ciclo. Inicialmente el análisis mapea el *parameter node* de nombre *this* con el argumento $I14$. El análisis mapea el *outside edge* $\langle this, cell, L12\rangle$ del método llamado con el *inside edge* $\langle I14, cell, L15\rangle$ previo a la llamada, y los nodos $L10$ y $L12$ con $L15$. Esta situación ilustra un elemento clave del análisis: el mapeo de *outside edges*(operaciones de lectura) con *inside edges*(operaciones de escritura) para detectar a qué nodos representan los *load nodes*.

La Figura 5.3(b) presenta el Points-To graph posterior a la llamada, donde los *outside edges* de $L12$ y $L10$ produjeron los *outside edge* de $L15$. La Sección 5.4 contiene una explicación formal del análisis inter-procedural. Por simplicidad el resultado que produjeron en el análisis los métodos `Trans.apply` y `List.add` sólo se incluye en el Points-To graph final (Figura 5.2).

La Figura 5.4 presenta el análisis inter-procedural para el llamado a `next()` en la segunda iteración del ciclo. El análisis procede de manera similar a la primera iteración con la diferencia que ahora se tiene una mayor cantidad de ejes y mapeos. La Figura 5.4(b) presenta el Points-To graph posterior a la llamada. Entre otros, $L12$ se mapea con $L13$ y el *outside edge* $\langle L12, next, L13\rangle$ del método llamado genera el *outside edge* $\langle L13, next, L13\rangle$. Este tipo de ejes se generan cuando se analizan o construyen estructuras recursivas. Luego se alcanza un punto fijo y el Points-To no se ve modificado.



(a) Mapeo Nodos

(b) Points-To Graph posterior a la llamada

Figura 5.4: Análisis inter-procedural para el llamado a `ListItr.next` de la segunda iteración del ciclo

Respecto a la rama del grafo relacionada con el retorno y los nodos que no poseen ejes, se construye de manera similar a la mencionada con la salvedad que los métodos llamados son `List.add()` y `Trans.apply`. La Figura 5.2 ilustra el resultado final del análisis del método.

5.3. Algoritmo Intra-procedural

El análisis de escape presentado en [SR05b, SR05a] es un análisis dataflow [NNH99] el cual parte del Points-To Graph:

$$\langle \emptyset, \emptyset, \{P_i \mapsto n^P_{m,i}\}_{0 \le i \le k-1}, \emptyset \rangle$$

Luego evoluciona al mismo usando la función de transferencia $Transf_{lb}$ presentada en el cuadro 5.1, donde $lb \in Label$. La función de transferencia $Transf_{lb}$ toma el Points-To graph $G = \langle I, O, L, E \rangle$ para el punto de programa anterior a lb y produce el Points-To graph para el punto de programa siguiente a lb.

Instrucción de label lb	$Transf_{lb}(\langle I, O, L, E\rangle)$
$lb : v_1 = v_2$	$\langle I, O, L[v_1 \mapsto L(v_2)], E\rangle$
$lb : v = new\ T$	$\langle I, O, L[v \mapsto \{n^I_{lb}\}], E\rangle$
$lb : v_1.f = v_2$	$\langle I \cup (L(v_1) \times \{f\} \times L(v_2)), O, L, E\rangle$
$lb : v_1[i] = v_2$	$\langle I \cup (L(v_1) \times \{[]\} \times L(v_2)), O, L, E\rangle$
$lb : C.f = v$	$\langle I, O, L, E \cup L(v)\rangle$
$lb : v_1 = v_2.f$	$process_load(G, v_1, v_2, f, lb)$
$lb : v_1 = v_2[i]$	$process_load(G, v_1, v_2, [], lb)$
$lb : v = C.f$	$\langle I, O, L[v \mapsto \{n_{GBL}\}], E\rangle$
$lb : if\ (...)$	$\langle I, O, L, E\rangle$ $(no\ se\ modifica)$
$lb : v_R = call\ m(v_1, ..., v_j)$	**Caso 1:** llamada analizable - Ver Sección 5.4
	Caso 2: llamada no analizable $\langle I, O, L[v_R \mapsto \{n_{GBL}\}], E \cup \bigcup_{i=0}^{j} L(v_i)\rangle$
$lb : return\ v$	$\langle I, O, L[v_{ret} \mapsto L(v)], E\rangle$

Cuadro 5.1: Función de transferencia de Rinard para las instrucciones relevantes

Además, en el cuadro 5.2 se presenta el proceso *process_load*. Sus argumentos son, en el orden que se mencionan, el Points-To graph previo al load ($G = \langle I, O, L, E\rangle$), la variable v_1 en la que se carga, la variable v_2 de la que se carga, el campo cargado f y el label lb de la instrucción LOAD «$v_1 = v_2.f$». Devuelve el Points-To graph posterior a la instrucción.

$$
\begin{array}{l}
process_load(\langle I, O, L, E\rangle, v_1, v_2, f, lb) = \\
\quad \textbf{let } A = \{n \in Node | \exists n_1 \in L(v_2), \langle n_1, f, n\rangle \in I\} \\
\quad\quad B = \{n \in L(v_2) | escape(G)(n)\} \textbf{ in} \\
\quad \textbf{if } \ B = \emptyset \\
\quad\quad \textbf{then } \ \langle I, O, L[v_1 \mapsto A], E\rangle \\
\quad\quad \textbf{else } \ \langle I, O \cup (B \times \{f\} \times \{n^L_{lb}\}), L[v_1 \mapsto (A \cup \{n^L_{lb}\})], E\rangle
\end{array}
$$

Cuadro 5.2: Procedimiento *process_load*.

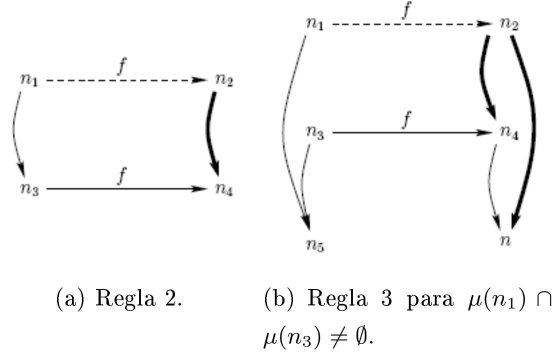
5.4. Algoritmo Inter-procedural

El análisis inter-procedural utiliza el Points-To graph G previo a realizar una llamada a un método *callee* junto con el resumen G_{callee} generado por este método para calcular uno correspondiente al punto de programa posterior a la llamada realizada. En el caso que existan varios posibles métodos llamados se realiza la unión de los grafos de cada uno de ellos, tratando a este caso de manera conservadora.

El análisis inter-procedural contiene tres pasos:

1. Se computa una relación μ' que mapea nodos de G_{callee} a nodos que aparecen en el grafo final, desambiguando la mayor cantidad posible de nodos pertenecientes a $PNodes \cup LNodes$.

2. Se utiliza el mapping de 1 para combinar G y G_{callee}. Cada nodo n perteneciente a G_{callee} es proyectado a través del mapping μ', es decir, intuitivamente n es



(a) Regla 2. (b) Regla 3 para $\mu(n_1) \cap \mu(n_3) \neq \emptyset$.

Figura 5.5: Reglas inter-procedural Salcianu-Rinard

reemplazado con los nodos de $\mu'(n)$.

3. Se simplifica el Points-To graph resultante removiendo *load nodes* redundantes y *outside edges* tal como se explica en [SR05b, SR05a].

5.4.1. Construcción del mapeo de nodos

Inicialmente se calcula un mapping μ que desambigüe la mayor cantidad de *parameter* y *load nodes* del método llamado *callee* que sean posibles. Sea $G = \langle I, O, L, E \rangle$ y $G_{callee} = \langle I_{callee}, O_{callee}, L_{callee}, E_{callee} \rangle$, se define μ como el menor punto fijo de las siguientes reglas:

1. $L(v_i) \subseteq \mu(n^P_{callee,i}) \forall i \in \{0, 1, ..., j\}$.

2. $$\frac{\langle n_1, f, n_2 \rangle \in O_{callee}, \langle n_3, f, n_4 \rangle \in I, n_3 \in \mu(n_1)}{n_4 \in \mu(n_2)} .$$

3. $$\frac{\begin{array}{c}\langle n_1, f, n_2 \rangle \in O_{callee}, \langle n_3, f, n_4 \rangle \in I_{callee}, \\ (\mu(n_1) \cup \{n_1\}) \cap (\mu(n_3) \cup \{n_3\}) \neq \emptyset, \\ (n_1 \neq n_3) \vee (n_1 \in LNode)\end{array}}{\mu(n_4) \cup (\{n_4\} \setminus PNode) \subseteq \mu(n_2)} .$$

La regla 1 mapea cada *parameter node*$n^P_{callee,i}$ con el nodo apuntado por v_i, el i-esimo argumento pasado a *callee*. Las reglas restantes extienden μ mapeando *outside edges* con *inside edges*.

La regla 2 maneja el caso en que el método llamado lee referencias creadas por el llamador. Mapea un *outside edge* $\langle n_1, f, n_2 \rangle \in O_{callee}$ del llamado con un *inside edge* $\langle n_3, f, n_4 \rangle \in I$ del llamador en el caso que n_1 podría representar a n_3, es decir, $n_3 \in \mu(n_1)$. Como n_1 podría ser n_3, el *outside edge* leido de n_1 podría ser el *inside edge* $\langle n_3, f, n_4 \rangle$ y el *load node* n_2 podría ser el nodo n_4. Por lo tanto el análisis mapea n_2 con n_4, es decir, $n_4 \in \mu(n_2)$. La Figura 5.5(a) ilustra la situación mencionada.

La regla 3 mapea un *outside edge* del método llamado con un *inside edge* del método llamado, es decir, del mismo contexto de análisis. La regla trata con el aliasing

presente en el contexto de llamada. Consideremos un *outside edge* $\langle n_1, f, n_2 \rangle \in O_{callee}$ y un *inside edge* $\langle n_3, f, n_4 \rangle \in I_{callee}$. La Figura 5.5(b) ilustra el caso donde n_1 y n_3 podrían representar al mismo nodo n_5. En este caso n_2 podría ser n_4, por lo tanto se fuerza $n_4 \in \mu(n_2)$. Además, como n_4 es un nodo del método llamado podría representar a otros nodos. Por lo tanto, n_2 podría representar no solo a n_4 sino a todos los nodos representados por n_4. Por lo tanto, se actualiza el mapping para representar $\mu(n_4) \subseteq \mu(n_2)$. De la misma forma n_1 podría representar a n_3 y n_3 a n_1.

Finalmente se calcula un mapping μ' extendiendo el mapping anterior de cada nodo que no sea *parameter node* a si mismo.

$$\forall n, \mu'(n) = \mu(n) \cup (\{n\} \setminus PNode)$$

μ' mapea nodos de G_{callee} a nodos que aparecen en el Points-To graph posterior a la llamada. Mientras que para los *inside nodes* representa la relación de identidad, los *parameter nodes* son ignorados. Además, los *load nodes* no son totalmente desambiguados en esta instancia.

5.4.2. Combinando los Points-To graph

Luego de calcular μ', el mismo se usa para combinar el Points-To graph G previo a la llamada con el G_{callee} calculado para el final del método *callee*. Se obtiene $G_2 = \langle I_2, O_2, L_2, E_2 \rangle$, válido si *callee* es llamado, de la siguiente forma:

$$\begin{gathered} I_2 = I \cup \bigcup_{\langle n_1, f, n_2 \in I_{callee} \rangle} \mu'(n_1) \times \{f\} \times \mu'(n_2) \\ O_2 = O \cup \bigcup_{\langle n, f, n^L \in O_{callee} \rangle} \mu'(n_) \times \{f\} \times \mu'(n^L) \\ L_2 = L[v_R \mapsto \mu'(L_{callee}(v_{ret}))] \\ E_2 = E \cup \mu'(E_{callee}) \end{gathered}$$

5.5. Extendiendo hacia k-sensitivo

El análisis hasta aquí descripto no utiliza información del contexto en que se produce una invocación a la hora de calcular los resúmenes. Esto se debe a que cómo los nodos sólo son identificados por el label donde son generados, sólo es posible distinguir objetos creados en diferentes líneas de un mismo método (i.e. el análisis es 0-sensitivo). Por lo tanto, no es posible distinguir nodos provenientes de llamadas al mismo método aún cuando las mismas se produzcan en contextos distintos. Esto puede llevar a una pérdida de precisión del análisis. Por ejemplo, si un método m hace dos invocaciones a un método m', en el cual los objetos generados por la primera invocación son temporales y los generados por la segunda son residuales, el análisis concluirá que todos son residuales. Esto se debe a que utiliza el mismo nodo para representar ambas invocaciones.

Consideremos el método *testMultiply* presentado en el Código de ejemplo 5.2.

```
0 Integer[] multiply(Integer[] v, int f)
    {
1   Integer[] res = new Integer[v.
        length];
2   for(int i=0;i<v.length;i++) {
3     Integer factor = new Integer(f);
4     res[i] = new Integer(v[i] *
          factor);
5   }
6   return res;
7 }
8
9
10 void testMultiply(Integer[] v1,
       Integer[] v2) {
11   Integer[] m1 = multiply(v1,2);
12   Integer[] m2 = multiply(v2,5);
13 }
```

Listing 5.2: Código de ejemplo para explicar k-sensitividad

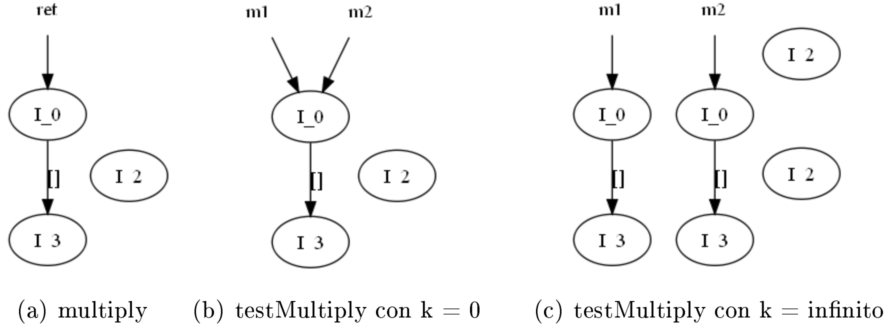
Se puede ver que existen dos objetos Integer[] definidos, uno por cada invocación a `multiply`. Sin embargo la Figura 5.6(b) presenta el resultado de ejecutar el análisis de escape donde se observa que sólo existe un *inside node* (I_0) que representa a ambos arreglos. El nodo $I2$ proviene del análisis de `multiply` pero no es relevante.

Como el análisis es 0-sensitivo, no es posible distinguir el arreglo retornado por la invocación en la línea 11 (apuntado por $m1$) de aquél retornado en la línea 12 (apuntado por $m2$). Para poder distinguirlos alcanza con modificar el análisis para que distinga llamadas producidas en diferentes líneas del mismo método, es decir para que sea 1-sensitivo.

Con el objetivo de lograr el máximo de precisión posible, se modifica el análisis de forma tal que a los nodos se les asocia una pila de tamaño k, buscando distinguir cadenas de hasta k llamadas, donde k puede ser infinito. Cada vez que se analiza una invocación, se agrega a la pila de los nodos que se agregan al grafo del llamador, el label de la instrucción de invocación. Si al momento de agregar algo en la pila la misma se encuentre llena, se elimina de la misma el elemento más antiguo. Esto da el efecto de k-sensitividad. De esta forma, los nodos del Points-To graph dejan de identificarse solamente por el label de la instrucción por la que son creados (ver Sección 5.3) y pasan a identificarse por el label y el contenido de la pila asociada a los mismos. Para poder lograr un análisis k-sensitivo alcanza con modificar levemente la componente inter-procedural del análisis de escape de [SR05b, SR05a]. Este análisis es iniciado cuando la componente intra-procedural se encuentra con una instrucción de tipo $cs : call\ m'(\bar{args})$. Previo al cálculo del mapping μ se realiza una copia del Points-To graph del *callee* donde a cada nodo del mismo se le realiza un *push* de cs en la pila asociada al mismo.

El análisis original de escape de [SR05b, SR05a] presentado anteriormente resulta equivalente a tener una pila de tamaño 0 para cada nodo.

En la Figura 5.6(c) se observa el resultado del análisis con k infinito. Notar que el resultado del análisis del ejemplo presentado es equivalente para cualquier $k \geq 1$



(a) multiply (b) testMultiply con k = 0 (c) testMultiply con k = infinito

Figura 5.6: Análisis testMultiply variando valor de k

5.6. Limitaciones

Una limitación existente es que el análisis no ofrece la posibilidad de detectar cuándo un objeto que escapa globalmente puede considerarse capturado. Si bien esto es razonable, pues un objeto que escapa globalmente es accesible de cualquier punto del programa y por tanto un análisis conservador y composicional debe asumir que escapa, a efectos del análisis de consumo se podría realizar algún tipo de análisis auxiliar para determinar cuando los objetos que escapan globalmente dejan de ser referenciados y de esta forma capturarlos.

La asignación a campos de un objeto siempre produce un weak update, es decir a la hora de actualizar una referencia no elimina los ejes existentes sino que simplemente agrega información. Se cree que es posible optimizar el análisis detectando configuraciones del heap donde el campo a actualizar referencie a un único nodo y en estos casos hacer un strong update, es decir eliminar el eje existente y reemplazarlo por los nuevos. Esto puede influir positivamente en la precisión del análisis.

Por otro lado, se utiliza una única etiqueta para relacionar arreglos con su contenido. Esto si bien favorece a que el análisis sea simple, como consecuencia se pierde precisión en cuanto a los objetos que efectivamente se acceden al indexar un arreglo.

5.7. Conclusiones

En este capítulo presentamos el análisis de escape de [SR05b, SR05a], sus limitaciones, y una adaptación del mismo para dar el efecto de k-sensitividad con el objetivo de una mayor precisión.

En el siguiente capítulo se da una instanciación del algoritmo descripto en el capítulo anterior. Para esto se define una representación para las expresiones paramétricas y las particiones de memoria, y se presenta una instanciación del algoritmo. A su vez se da una implementación para las herramientas auxiliares presentadas a lo largo del capítulo anterior.

Capítulo 6

Calculando efectivamente requerimientos de memoria

En el capítulo anterior presentamos el análisis de escape de [SR05b, SR05a], sus limitaciones, y una adaptación del mismo para dar el efecto de k-sensitividad con el objetivo de una mayor precisión.

En este capítulo se presenta una instanciación del algoritmo descripto en el Capítulo 4. La misma queda definida por la implementación de las herramientas auxiliares necesarias para el algoritmo (CT, SA y EA), y la forma en que se representan las expresiones paramétricas y las particiones de memoria. Además se presenta una problemática surgida en los casos que se presenten llamadas polimórficas y una serie de estrategias para mejorar las cotas obtenidas.

A lo largo de este capítulo se utiliza el Código de ejemplo 5.1.

6.1. Expresiones Paramétricas

Como se mencionó anteriormente, una expresión paramétrica $e \in E : \bar{X} \mapsto \bar{K} \mapsto N$ es una función de valuaciones de parámetros relevantes e instanciaciones de tipos (e.g. sus tamaños) en números naturales. Además, recordemos que por simplicidad se asume $\forall k \in \bar{k}$, el tamaño de $k = 1$, haciendo que las expresiones queden reducidas a una función de valuaciones en números naturales $e \in E : \bar{X} \mapsto N$.

Para mejorar la precisión del análisis se representa el consumo de memoria por casos. A su vez, la calculadora `iscc` utilizada trabaja con polinomios (Ver Sección 7.3). Por lo tanto, se decidió representar una expresión paramétrica como una función por casos, donde cada caso es un polinomio en $\bar{X}$ y cada parámetro perteneciente a $\bar{x}$ es un valor entero (ver Sección 2.3). Esto restringe las expresiones de consumo con las que se trabaja haciendo que sólo sea posible calcular consumo polinomial (para consumos mayores dará infinito o será necesaria alguna codificación). Como la calculadora `iscc` sólo puede operar con restricciones lineales, los predicados que

marcan los casos se encuentran limitados a este tipo de restricciones.

Luego, las expresiones quedan definidas como:

$$e = [Pol_i : \phi_i] \;\; i : \;\; 1...n$$

Y se evalúan de la siguiente forma:

$$e(\bar{x}) = \begin{cases} Pol_i(\bar{x}) & si \; \phi_i(\bar{x}) \\ 0 & en \; otro \; caso \end{cases}$$

Notar que dos predicados ϕ no pueden ser verdaderos simultáneamente para garantizar la elección de un único caso y que cada polinomio tiene asociado un predicado lineal ϕ en $\bar{X}$ que indica cuándo debe ser seleccionado. A la hora de evaluar una expresión, estos predicados permiten elegir qué caso evaluar. Por otro lado, se decidió representar con la condición $True$ el caso en donde la función es completa y tiene un único caso.

Ejemplo El consumo del método `map` presentado en el Código de ejemplo 5.1 que podía ser resumido como $\texttt{map}.Tmp = 3$ y $\texttt{map}.Rsd(ret) = 5 * list.size + 1$ según lo desarrollado en el Capítulo 3 puede expresarse como:

$$\texttt{map}.Tmp = \{3 : True\} \; y \; \texttt{map}.Rsd(ret) = \{5 * list.size + 1 : True\}$$

Además, podría expresarse de forma más precisa como:

$$\texttt{map}.Tmp = \{3 : list.size > 0; 1 : otro \; caso\}$$

$$\texttt{map}.Rsd(ret) = \{5 * list.size + 1 : list.size > 0; 1 : otro \; caso\}$$

Esta expresión denota el hecho que con la lista vacía no se ejecutará el cuerpo del ciclo. Por lo tanto el residual sólo estará formado por la lista *list* y el temporal provendrá únicamente de la ejecución del método `list.iterator()`. Por simplicidad, de aquí en adelante no se indicará el predicado asociado a un polinomio cuando este sea $True$.

6.2. Particiones de memoria

Como se mencionó anteriormente, un *resumen de consumo de memoria* para un método m es la tupla $S_m = \langle \bar{p}, \bar{x}, \bar{\gamma}, \bar{k}, Tmp, Rsd \rangle$, donde Rsd es una función que dada una partición del *heap* retorna una expresión paramétrica que representa el consumo de los objetos incluidos en esa partición. Sea $G_m = \langle I_m, O_m, L_m, E_m \rangle$ el Points-To graph resultante del análisis de escape de m, $ILNode_m = \{n | n \in INode_m \cup LNode_m\}$ el conjunto de nodos de G_m excluyendo a los parámetros

($PNode_m$) y a los nodos globales, y $escape(G_m)(n)$ una función que dado G_m determina si el nodo n escapa a G_m (ver Sección 5.1). Se decide utilizar como granularidad una partición por cada nodo que escapa a G_m, es decir que existirá una partición asociada a cada nodo del conjunto $\{n \mid n \in ILNode_m \wedge escape(G_m)(n)\}$.

Luego, se decide modelar un resumen S_m para esta implementación como:

- $\bar{p} : \bar{P}$ es la lista $\bar{p_m} = p_1, ..., p_n$ de parámetros formales de m.
- $\bar{x} : \bar{X}$ es la lista $\bar{x_m} = x_1, ..., x_k$ de parámetros relevantes de m. Recordar que los parámetros relevantes son valores enteros.
- $\bar{\gamma} : \bar{\Gamma}$ es la lista $\bar{\gamma_m} = \gamma_1, ..., \gamma_k$ de funciones las cuales relacionan los parámetros formales del método con los parámetros relevantes del mismo.
- $Tmp : \mathbb{E}[\bar{X}]$ es una expresión que representa el consumo de los nodos que no escapan a G_m, es decir los nodos del conjunto $\{n \mid n \in ILNode_m \wedge \neg(escape(G_m)(n))\}$.
- $Rsd : \mathsf{HP} \mapsto \mathbb{E}[\bar{X}]$ es una función que toma como dominio las particiones asociadas a los nodos que escapan a G_m, es decir $\mathsf{HP} = \{partition(n) \mid n \in ILNode_m \wedge escape(G_m)(n)\}$, donde $partition(n)$ es una función que devuelve la partición asociada a un nodo. A través de la función Rsd, cada partición de HP tiene asociada una expresión paramétrica que representa su consumo de memoria.

Es importante notar que tomar como granularidad los nodos del análisis de escape permite aprovechar al máximo la precisión del análisis. De esta forma, los *inside nodes* tendrán asociados particiones que aporten consumo, mientras que los *load nodes* tendrán asociados particiones sin consumo pues representan información de lecturas. Si bien estas particiones no aportan consumo, forman parte del resumen para poder tener una representación gráfica del las relaciones entre las particiones. Quitarlas imposibilitaría graficar el resumen como un grafo, característica que resulta útil para entender los resultados del análisis.

6.3. Implementación de los componentes

A continuación se presenta la implementación de las herramientas auxiliares del algoritmo presentado.

6.3.1. Implementación del CT

Como se definió anteriormente, el *CT* es una herramienta capaz de contar la cantidad de veces que una instrucción es visitada mediante la operación $count : Methods \times \bar{P} \times \bar{\Gamma} \times Labels \mapsto \mathbb{E}[\bar{X}]$.

Esta operación es implementada siguiendo la aproximación presentada en [BFGY08, RG09] la cual requiere que existan invariantes lineales numéricos (las variables deben representar enteros) para describir el espacio de iteración en el que es ejecutada una instrucción. A los efectos de la implementación del CT se asume que los invariantes son inductivos, esto quiere decir que estarán expresados sólo en términos de variables que influyen en la cantidad de iteraciones en que se ejecuta una instrucción (i.e. variables inductivas). Este requerimiento dificulta la inferencia automática de invariantes pero facilita el procesamiento de los mismos por parte del CT puesto que no es necesario eliminar las variables no inductivas previo al uso de los invariantes.

Dentro del conjunto de variables inductivas que forman parte de un invariante, es posible hacer una división entre aquellas variables que se asumen constantes (i.e. tienen un valor fijo en cada iteración), y aquellas que están libres (i.e. pueden variar en cada iteración del ciclo). Como se busca obtener un resultado en términos de los parámetros relevantes del método, las variables que se asumen fijas serán los parámetros relevantes.

De esta forma, para un punto del programa con label l, se obtiene un invariante lineal $I_l(\bar{x_m}, \bar{w})$ que define el espacio de iteración asociado a la instrucción l, donde $\bar{x_m}$ serán las variables que se asumen fijas (i.e. los parámetros relevantes) y $\bar{w}$ aquellas que se encuentran libres.

Como las variables son enteras, la cantidad de visitas a una instrucción puede ser calculada obteniendo la cantidad de valuaciones para las variables libres $\bar{w}$ que satisfacen al invariante I_l, en otras palabras, contando la cantidad de soluciones enteras del invariante. Por lo tanto, la operación de *count* se resuelve devolviendo una expresión paramétrica que represente dicho valor, es decir:

$$count(m, \bar{p_m}, \bar{\gamma_m}, l) = \sharp\{\bar{w} \in \bar{\mathbb{N}} \mid I_l(\bar{\gamma_m}(\bar{p_m}), \bar{w})\}$$

Es importante notar que el invariante esta expresado en términos de los parámetros relevantes del método m mientras que la operación *count* recibe como argumento los parámetros formales. Por esta razón, es necesario hacer uso del mapping $\bar{\gamma_m}$ para obtener los parámetros relevantes a partir de los formales.

Ejemplo Para determinar la cantidad de veces que fue visitada la instrucción *Object o = transform.apply(it.next());* del método `map` (línea 51), considerando 51 como el label de la misma, se puede definir el invariante

$$I_{51}(\{list.size\}, \{i\}) := 0 \leq i < list.size$$

que representa el espacio de iteración en que la misma es ejecutada. Las soluciones a I_{51} son $i = 0, i = 1, ..., i = list.size - 1$. Por lo tanto existen $list.size$

valuaciones que satisfacen el invariante, lo que significa que la instrucción *Object o = transform.apply(it.next());* es ejecutada $list.size$ veces.

Formalmente, si se considera

$$\bar{\gamma} = [\gamma_{list.size}(list, transform) = list.size]$$

la operación *count* para esta instrucción se resuelve de la siguiente manera:

$$count(map, [list, transform], \bar{\gamma}, 51) =$$
$$= \sharp\{i \in \mathbb{N} \mid I_{51}(\bar{\gamma}([list, transform]), \{i\})\} =$$
$$= \sharp\{i \in \mathbb{N} \mid 0 \leq i < list.size\} = list.size$$

Para contar la cantidad de soluciones a un invariante lineal numérico se utiliza el operador "*card*" de la calculadora `iscc` definida en la librería de conjunto de enteros [Ver07, Ver10]. La misma requiere expresar los invariantes con notación $\{[< variables\ constantes >] \mapsto [< variables\ libres >] :< condiciones >\}$. Por lo tanto, para calcular la cantidad de veces que la instrucción fue ejecutada, basta con pedirle a la calculadora `iscc` que ejecute el comando $card\{[list.size] \mapsto [i] : 0 \leq i < list.size\}$; para obtener el resultado $\{list.size : list.size >= 0\}$.

Para el caso de las instrucciones ejecutadas una única vez, como la instrucción *List res = new List();* (línea 49), es posible definir el invariante $I_{49}(\emptyset, i) := 1 = i$ o bien asumir que un invariante vacio $I_{49}(\emptyset, \emptyset) := \emptyset$ equivale al polinomio constante 1.

El CT definido cumple con los requerimientos del algoritmo, permitiendo calcular la cantidad de visitas a cualquier instrucción del programa. Notar que la utilización de invariantes lineales, inductivos y numéricos restringe el espacio de programas sobre el cual es posible calcular la cantidad de visitas a instrucciones. Si bien esto parece una limitación importante, en la práctica este enfoque permite trabajar con una gran cantidad de programas. Por ejemplo, permite trabajar con cualquier programa que haga un uso intensivo de iteraciones, inclusive si son anidadas.

6.3.2. Implementación del SA

Como se definió anteriormente, el SA es una calculadora simbólica capaz de operar con las expresiones paramétricas y debe dar soporte a las siguientes operaciones:

- *summate*: $\mathbb{E}[\bar{X}'] \times \bar{\Gamma}' \times A\bar{r}gs \times \bar{P} \times \bar{\Gamma} \times Labels \mapsto \mathbb{E}[\bar{X}]$
- *maximize*: $\mathbb{E}[\bar{X}'] \times \bar{\Gamma}' \times A\bar{r}gs \times \bar{P} \times \bar{\Gamma} \times Labels \mapsto \mathbb{E}[\bar{X}]$
- *add*: $\mathbb{E}[\bar{X}] \times \mathbb{E}[\bar{X}] \mapsto \mathbb{E}[\bar{X}]$
- *join*: $\mathbb{E}[\bar{X}] \times \mathbb{E}[\bar{X}] \mapsto \mathbb{E}[\bar{X}]$

Operaciones summate y maximize: Para las operaciones *summate* y *maximize* se utiliza un enfoque similar al utilizado en la implementación del CT. Es decir, para resolver estas operaciones se trabaja con invariantes lineales numéricos e inductivos. Como el resultado debe estar expresado en términos de los parámetros relevantes del método m que realiza la invocación, además de expresar la cantidad de visitas a la instrucción de `call` que es visitada, el invariante relaciona los argumentos en los que se evalúa la expresión dada como parámetro con los parámetros relevantes de m.

De esta forma, para un punto del programa con label cs donde ocurre una invocación, se obtiene un invariante lineal con la forma:

$$I_{cs}(\bar{\gamma_m}(\bar{p_m}), \bar{\gamma_{m'}}(\bar{args}) \cup \bar{w})$$

El mismo define el espacio de iteración asociado a la instrucción cs, el binding entre argumentos y parámetros del método invocado y la relación entre argumentos y parámetros relevantes del método al que pertenece cs. El conjunto de variables libres $\bar{w}$ contiene aquellas variables inductivas que no son parámetros ni argumentos, necesarias para poder expresar correctamente el invariante.

La operaciones *summate* y *maximize* deben operar con los valores que toma la expresión e a lo largo de cada visita a la instrucción asociada al punto de programa cs, siendo e y cs dos de los argumentos de las operaciones. La primer operación acumula estos valores y la segunda obtiene el máximo de los mismos. Los invariantes fueron definidos de forma tal que permiten obtener los valores en los cuales evaluar la expresión e en cada visita. Una valuación para las variables libres del invariante que sea solución del mismo, provee valores para los argumentos en los que debe evaluarse la expresión. Utilizando el conjunto de todas las soluciones del invariante, es posible obtener todos los valores que toma la expresión e en cada visita y así resolver las operaciones de *summate* y *maximize*.

De esta forma las operaciones pueden ser expresadas formalmente como:

$$summate(e, \bar{\gamma_{m'}}, \bar{args}, \bar{p_m}, \bar{\gamma_m}, cs) = \text{Acumular } e(\bar{x'_{args}}) \text{ sujeto a } I_{cs}(\bar{x_p}, \bar{x'_{args}} \cup \bar{w})$$

$$maximize(e, \bar{\gamma_{m'}}, \bar{args}, \bar{p_m}, \bar{\gamma_m}, cs) = \text{Maximizar } e(\bar{x'_{args}}) \text{ sujeto a } I_{cs}(\bar{x_p}, \bar{x'_{args}} \cup \bar{w})$$

donde $\bar{x_p} = \bar{\gamma_m}(\bar{p_m})$ y $\bar{x'_{args}} = \bar{\gamma_{m'}}(\bar{args})$.

Ejemplo Consideremos la llamada al método `safeMap` realizada en la línea 72 del método *test* presentado en el Código de ejemplo 5.1. Se utiliza $\$p.list.size$ para predicar acerca del parámetro del método invocado, con el fin de evitar colisiones entre éste y los parámetros del llamador. El invariante necesario para evaluar esta invocación es: $I_{72}(\{list.elems.maxSize, list.size\}, \{\$p.list.size, i\}) := 1 \leq i \leq list.size \text{ } and \text{ } \$p.list.size = list.elems.maxSize$.

Se puede ver que el mismo está compuesto por una primer restricción que describe la cantidad de veces que la invocación será ejecutada, es decir tantas veces como elementos tenga la lista de parámetro que recibe `test`, y otra que describe la relación entre los parámetros del método invocado y los argumentos de la invocación. Como se mencionó en el Capítulo 3, se utiliza el largo máximo de los elementos de la lista como cota ($list.elems.maxSize$).

Según la explicación dada en el Capítulo 3, el resumen del método `safeMap` puede ser expresado como `safeMap`.$Tmp = list.size + 4$ y `safeMap`.$Rsd(ret) = 5 * list.size + 1$.

Para simplificar el largo de la expresión del ejemplo se utiliza s en lugar de $list$, e en lugar de $elements$, mS en lugar de maxSize. Además para facilitar la comprensión se utiliza $expr$ en lugar de $5 * \$p.s.size + 1$, $expr2$ en lugar de $\$p.s.size + 4$, $\bar{\gamma_{m'}}$ en lugar de $[\gamma_{s.size}(s, transform) = s.size]$ y $\bar{\gamma_m}$ en lugar de $[\gamma_{s.size}(s, transform) = s.size, \gamma_{s.size}(s, transform) = s.e.mS]$. En base a esto, para calcular los efectos de la invocación a `safeMap` en el método `test` se realiza:

$$summate(expr, \bar{\gamma_{m'}}, [it.next, transform], [list, transform], \bar{\gamma_m}, test.l_{72}) =$$

$$= Acumular\ expr(s.e.mS)\ sujeto\ a\ \{1 \leq i \leq s.size\} =$$

$$= \{(1 + 5 * s.e.mS) * s.size : s.size >= 1\}$$

$$maximize(expr2, \bar{\gamma_{m'}}, [it.next, transform], [list, transform], \bar{\gamma_m}, test.l_{72}) =$$

$$= Maximizar\ expr2(s.e.mS)\ sujeto\ a\ \{1 \leq i \leq s.size\} =$$

$$= \{s.e.mS + 4 : s.size >= 1\}$$

La implementación de las operaciones de Acumular y Maximizar fueron resueltas utilizando los operadores "*sum*" y "*ub*" de la calculadora `iscc` [Ver07, Ver10] (Ver Sección 7.3). La misma requiere combinar en un único argumento de los operadores tanto la expresión como el invariante, utilizando la notación: $\{[[< variables\ constantes >] \mapsto [< variables\ libres >]] \mapsto < expresion >:< condiciones >\}$. El binding entre parámetros de la expresión y los argumentos en que debe ser evaluada forman parte de las condiciones. Por lo tanto, los ejemplos anteriores pueden ser resueltos mediante la calculadora ejecutando las siguientes operaciones respectivamente:

$$sum\ \{[[s.e.mS, s.size] \mapsto [\$p.s.size, i]] \mapsto 5 * \$p.s.size + 1 :$$

$$1 \leq i \leq s.size\ and\ \$p.s.size = s.e.mS\}$$

$$ub\ \{[[s.e.mS, s.size] \mapsto [\$p.s.size, i]] \mapsto \$p.s.size + 4 :$$
$$1 \leq i \leq s.size\ and\ \$p.s.size = s.e.mS\}$$

Operación add: La operación *add* fue implementada utilizando el operador "+" de `iscc` . El mismo opera sobre el conjunto de polinomios acumulando los valores para los conjuntos de restricciones que se intersequen. Es una operación compleja de resolver ya que la misma debe detectar los dominios que se intersecan.

Ejemplo 1

$$\{n : n > 5\} + \{n + 1 : n > 0\} = \{2 * n + 1 : n > 5; n + 1 : n > 0\ \&\&\ n <= 4\}$$

Ejemplo 2

$$\{n : n > 5; 4 * n : n <= 4\} + \{n + 1 : n > 0\} =$$
$$= \{2 * n + 1 : n > 5; 5 * n + 1 : n > 0\ \&\&\ n <= 4; 4 * n : n <= 0\}$$

En el primero ejemplo se puede ver como el resultado obtenido queda dividido en 2 partes, en donde la primera de ellas representa la suma de ambos polinomios ya que se interseca su conjunto de restricciones, y la segunda es simplemente el segundo polinomio. En el segundo ejemplo se puede ver de forma más clara que la operación resulta compleja de resolver, ya que según los polinomios de entrada y su conjunto de restricciones, la cantidad de restricciones del resultado puede aumentar.

Operación join: Para el operador *join* se analizaron distintas alternativas basadas en operadores existentes en `iscc` . Como se mencionó anteriormente, el cálculo del máximo entre polinomios es un problema dificil de resolver. Por ejemplo, la calculadora no es capaz de determinar que el resultado de $join(n + 5, n^2)$ es $n^2 : n \geq 3; n + 5 < 3$. Además, existe una complejidad inherente al problema ya que $join(n, n^2)$ no tiene un único polinomio como solución cuando $n = 1$ y la calculadora en estas situaciones determina que ambos polinomios son el máximo. El problema adquiere una complejidad mayor al considerar casos que involucren más variables.

Si bien la calculadora no puede dar una solución exacta para el problema de maximización de polinomios, cuenta con un conjunto de operadores que permiten abordar el problema retornando valores aproximados.

Una primer alternativa consiste en utilizar el operador "." el cual restringe dos polinomios a una expresión paramétrica con un nuevo dominio. El problema con esta operación es que, dependiendo de los parámetros, el resultado puede ser costoso de calcular en términos de performance. Además para esta operación la calculadora recibe y retorna candidatos en formato $max(candidato_1, candidato_2, candidato_3, ...)_i : \phi_i$. A partir de esta característica es

necesario agregar a los candidatos antes de realizar la operación el formato $max()$ en caso que no lo posean, y será posible quitárselo en caso que el resultado de la misma sea de la forma $max(candidato_1)_i : \phi_i$.

Ejemplo

$$join(\{n : n \geq 2\}, \{n^2 : n \geq 2\}) = \{max(n) : n \geq 2\}.\{max(n^2) : n \geq 2\} =$$

$$= \{max(n^2) : n \geq 2\} = \{n^2 : n \geq 2\}$$

Ejemplo 2

$$join(\{n : n \geq 0\}, \{n^2 : n \geq 0\}) = \{max(n) : n \geq 0\}.\{max(n^2) : n \geq 0\} =$$

$$= \{max(n, n^2) : n \geq 0\}$$

En el primer ejemplo se puede ver la existencia de casos donde el operador retorna un único candidato el cual efectivamente es el máximo entre los polinomios sobre los que se aplica la restricción. En el segundo ejemplo se puede ver que modificando levemente las restricciones de los polinomios, el operador ya no puede descartar el polinomio n y lo incluye en la lista de candidatos retornados.

Otra opción posible para resolver la operación es devolver la suma ("+") de los argumentos.

Ejemplo

$$join(\{n : n \geq 2\}, \{n^2 : n \geq 2\}) = \{n^2 + n : n \geq 2\}$$

$$join(\{n : n \geq 0\}, \{n^2 : n \geq 0\}) = \{n^2 + n : n \geq 0\}$$

En el ejemplo puede verse que el resultado obtenido no es una buena cota ya que sobreaproxima al valor real.

Estas dos posibilidades definen dos estrategias para comparar expresiones paramétricas: *lazy* y *add*. La ventaja de la estrategia lazy es que permite trabajar con los candidatos para descartarlos cuando una nueva restricción agregue suficiente información como para hacerlo. De ahí el nombre *lazy*. A diferencia de *add*, trabajar con candidatos puede impactar en la performance.

Ejemplo Lazy

$$join(\{m : m > 0 \;\&\&\; n \geq 0\}, \{m * n : m > 0 \;\&\&\; n \geq 0\}) =$$

$$= \{max(m) : m > 0 \;\&\&\; n \geq 0\}.\{max(m * n) : m > 0 \;\&\&\; n \geq 0\} =$$

$$= \{max(m, n * m) : m > 0 \;\&\&\; n \geq 0\}$$

$$join(\{max(m, n*m) : m > 0 \ \&\& \ n \geq 0\}, \{m+n : m > 0 \ \&\& \ n \geq 0\}) =$$

$$= \{max(m, n*m) : m > 0 \ \&\& \ n \geq 0\}.\{max(m+n) : m > 0 \ \&\& \ n \geq 0\} =$$

$$= \{max(n*m, n+m) : m > 0 \ \&\& \ n \geq 0\}$$

Ejemplo Add

$$join(\{m : m > 0 \ \&\& \ n \geq 0\}, \{m*n : n > 0 \ \&\& \ n \geq 0\}) =$$

$$= \{m + m*n : m > 0 \ \&\& \ n \geq 0\}$$

$$join(\{(m + n*m) : m > 0 \ \&\& \ n \geq 0\}, \{m+n : n > 0 \ \&\& \ n \geq 0\}) =$$

$$= \{n + 2*m + n*m) : m > 0 \ \&\& \ n \geq 0\}$$

Los ejemplos presentados muestran una primer aplicación de la operación *join* sobre un conjunto de candidatos y una segunda aplicación sobre el resultado de la primera y un nuevo candidato. Notar que al primer candidato se le incluyo una restricción sobre n a pesar que no es parte del candidato con el objetivo de simplificar el dominio del resultado. Esto se debe a que la calculadora requiere que se indique en todos los candidatos si son, por ejemplo, de la forma $n >= 0$ para evitar tener un conjunto de restricciones sobre $n < 0$.

En el primer ejemplo se muestra cómo la estrategia *lazy* permite descartar candidatos, siendo que para la segunda aplicación del operador *join* fue posible descartar el candidato m y obteniendo por lo tanto un resultado más preciso.

El segundo ejemplo muestra cómo la estrategia *add* con sucesivas aplicaciones del operador retorna un valor que sobreaproxima cada vez más a la cota obtenida por *lazy*. Por otro lado se observa que esta estrategia trabaja siempre con expresiones que no poseen candidatos, lo que deriva en una mejor performance a la hora de resolver la operación.

De esta forma, puede optarse por cualquiera de las estrategias en función de la precisión y performance deseada. En el caso de esta implementación se opto por buscar la mejor precisión posible, es decir por la utilización del operador ".“ y la estrategia *lazy*.

6.3.3. Implementación del EA

El *EA* se define de forma tal que puede resolver sus operaciones utilizando los resúmenes de escape que se asumen previamente calculados. Durante la definición de las operaciones se utiliza la función $escape(G_m)(n)$ que dado el Points-To graph G_m resultante del análisis de escape de un método m determina si el nodo n escapa a G_m (ver Sección 5.1).

Operación getPartition: La operación $getPartition$: $Labels \mapsto \mathsf{HP} \cup \mathsf{Temporal}$, que determina en qué partición es asignado un objeto creado localmente (instrucción `new`), es implementada evaluando si el nodo del resumen de escape asociado a la instrucción escapa o no. En caso de escapar, se retorna la partición residual asociada al nodo. La partición debe existir pues HP está definida como las particiones asociadas a los nodos del resumen de escape que escapan. En caso que el nodo sea capturado se retorna la partición Temporal.

Sea $as \in Labels$ el label asociado a una instrucción `new`, m el método al que pertenece la instrucción, G_m el Points-To graph resultante del análisis de escape, $node(G, as)$ una función que permite obtener el nodo asociado a una instrucción `new` y $partition(n)$ una función que permite obtener la partición asociada a un nodo. La operación $getPartition$: $Labels \mapsto \mathsf{HP} \cup \mathsf{Temporal}$ puede expresarse como

$$getPartition(as) = \begin{cases} partition(node(G_m, as)) & si\ escape(G_m)(node(G_m, as)) \\ \mathsf{Temporal} & en\ otro\ caso \end{cases}$$

Ejemplo Consideremos el método `List.Add` presentado en el Código de ejemplo 5.1 (líneas 2 a 4). Tomemos la instrucción `new` de la línea 3 siendo 3 el *label* de la misma. Sea $G_{list.add}$ el Points-To graph obtenido del análisis de escape de `List.add`. El operador $getPartition$ para el label 3 se resuelve de la siguiente forma:

$$getPartition(3) = \begin{cases} partition(node(G_{list.add}, 3)) & si\ escape(G_{list.add})(node(G_{list.add}, 3)) \\ \mathsf{Temporal} & en\ otro\ caso \end{cases}$$

En la Figura 5.1(e) es posible observar el Points-To graph del método `List.add`. Se puede observar que el nodo asociado a 3 es $I4$. Además, se puede ver que el mismo escapa al método $List.add$. Por lo tanto:

$$getPartition(3) = \begin{cases} partition(I4) & si\ escape(G_{list.add})(I4) \\ \mathsf{Temporal} & en\ otro\ caso \end{cases} = partition(I4)$$

Ejemplo Consideremos el método `Trans.apply` presentado en el Código de ejemplo 5.1 (líneas 41 a 45) y tomemos la instrucción `new` de la línea 43 siendo 43 el *label* de la misma. Sea $G_{Trans.apply}$ el Points-To graph obtenido del análisis de escape de `Trans.apply` e $I6$ el nodo asociado a la instrucción. En la Figura 5.1(h) se puede ver que el mismo es capturado por el método `Trans.apply`. Por lo tanto el operador $getPartition$ para el label 43 se resuelve de la siguiente forma:

$$getPartition(43) = \begin{cases} partition(I6) & si\ escape(G_{Trans.apply})(I6) \\ \mathsf{Temporal} & en\ otro\ caso \end{cases} = \mathsf{Temporal}$$

Operación bindPartition: En el caso de la operación $bindPartition$: $\mathsf{HP} \times \bar{Args} \times \bar{P} \times Labels \mapsto \mathsf{HP}$, para determinar el destino de la partición $hp'_{m'}$ de un método m' invocado, alcanza con observar qué ocurre con el nodo $n_{hp'}$ asociado a $hp'_{m'}$ durante el análisis de escape. Según las reglas definidas en el Capítulo 5, $n_{hp'}$ debe formar parte del resumen del método llamador m. De esta forma, si $n_{hp'}$ escapa a m se devuelve la partición residual hp_m asociada a $n_{hp'}$ en m. Caso contrario, se devuelve la partición Temporal de m.

Sea $cs \in Labels$ el label de una instrucción `call`, m el método al que pertenece la instrucción, G_m el Points-To graph resultante del análisis de escape, $toNode(hp)$ una función que dado una partición hp devuelve el nodo de escape asociado a ella, $pushCont(n, cs)$ una función que dado un nodo de escape n retorna un nodo n' el cual se obtiene de agregar en la pila de contextos de n la invocación asociada a cs (ver Sección 5.5) y $partition(n)$ una función que permite obtener la partición asociada a un nodo. La operación $bindPartition$: $\mathsf{HP} \times \bar{Args} \times \bar{P} \times Labels \mapsto \mathsf{HP}$ puede expresarse como:

$$bindPartition(hp, \bar{args}, \bar{p_m}, cs) =$$

$$= \begin{cases} partition(pushCont(toNode(hp), cs)) & si\ escape(G_m)(pushCont(toNode(hp), cs)) \\ \mathsf{Temporal} & en\ otro\ caso \end{cases}$$

Notar que en la expresión no se ha hecho uso de los parámetros $\bar{args}$ y $\bar{p_m}$. Esto ocurre porque el vínculo entre parámetros del método invocado y argumentos provistos por el método invocador ya fue realizado durante el análisis de escape.

Ejemplo Consideremos el método `testMultiply` (ver Código de ejemplo 5.2). En la línea 11 se produce una invocación al método `multiply`, por lo tanto habrá que efectuar la parte inter-procedural del algoritmo. Si observamos los resultados del análisis de escape para el método `multiply` (ver Figura 5.6) es posible observar que en algún momento el algoritmo debe procesar la partición de memoria hp_{I2} asociada al nodo $I2$. Sea 9 el *label* de la instrucción de invocación, G_{tM} el Points-To obtenido como resultado del análisis de escape para el método *testMultiply*, $v1.length$ el argumento y $v.length$ el parámetro. El operador $bindPartition$ se resuelve de la siguiente forma:

$$bindPartition(hp_{I2}, v1.length, v.length, 9) =$$

$$= \begin{cases} partition(pushCont(toNode(hp_{I2}), 9)) & si\ escape(G_{tM})(pushCont(toNode(hp_{I2}), 9)) \\ \mathsf{Temporal} & en\ otro\ caso \end{cases} =$$

$$= partition(pushCont(toNode(hp_{I2}), 9))$$

6.4. Polimorfismo

El objetivo de esta sección es evaluar las características del algoritmo y el análisis definidos previamente respecto al polimorfismo y abordarlos con distintas estrategias con el objetivo de intentar mejorar los resultados para la precisión y la performance.

Se entiende como polimorfismo al hecho de no poder determinar estáticamente la *única* implementación que será ejecutada producto de una invocación virtual. Esto puede ocurrir, por ejemplo, cuando el argumento de un método es del tipo de una interfaz o una clase que posee extensiones.

En el Código de ejemplo 6.1 puede observarse que la invocación dentro de `map` al método `apply` (línea 24) es polimórfica ya que no es posible determinar estáticamente si el método que se ejecutará será `Trans.apply` o `Trans2.apply`.

```
0
1 class Trans {
2   Object apply(Object object) {
3     Date date = new Date();
4     Random random = new Random(date.
          getTime());
5     int value = ((Integer) object).
          intValue();
6     return new Integer( value * random
          .nextInt());
7   }
8 }
9
10 class Trans2 extends Trans {
11   Object apply(Object object) {
12     Date date = new Date();
13     Random random = new Random(date.
          getTime());
14     int value = ((Integer) object).
          intValue()
15     return new Integer[]{
16       new Integer(value),
17       new Integer(value * random.nextInt()
          )};
18   }
19 }
20
21 List map(List list, Trans transform)
     {
22   List res = new List();
23   for (Iterator it = list.iterator();
          it.hasNext();) {
24     Object o = transform.apply(it.
          next());
25     res.add(o);
26   }
27   return res;
28 }
```

Listing 6.1: Código de ejemplo para una invocación polimórfica.

El algoritmo 4.2 definido previamente contempla este problema ya que uno de sus objetivos es mejorar las cotas en presencia de polimorfismo. Para esto busca una cota para la partición temporal y las residuales que permita asegurar la correcta ejecución de cualquier implementación candidata para la invocación virtual. En otras palabras, se contempla la posibilidad de que cualquier candidato sea ejecutado.

Como se mencionó en la Sección 4.1, la cota para el temporal se logra calculando el supremo entre los tamaños de las particiones temporales de todas las implementaciones candidatas. En el caso del consumo residual, para cada partición hp_m del método resumido m, el algoritmo toma el supremo del residual aportado por cada implementación m_i de una invocación virtual, donde el aporte proviene de la suma del residual de todas las particiones de m_i que son vinculadas con hp_m a través del EA.

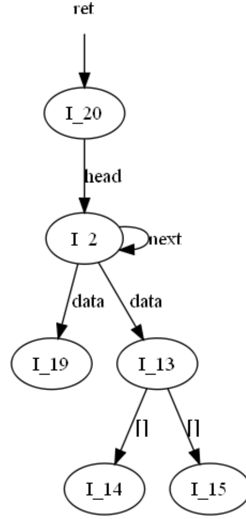


Figura 6.1: Resultado análisis escape para el método `map`

Por otro lado, el análisis de escape de [SR05b, SR05a] implementado aborda el problema de polimorfismo realizando la unión de Points-To graph obtenidos al considerar por separado la invocación a cada posible implementación. Al realizar la unión de Points-To graph el resultado contendrá nodos de varias implementaciones, rompiendo el invariante deseable del resumen de consumo el cual dice que el consumo residual de un método puede ser calculado como $\sum_{hp_m \in Dom(S_m.Rsd)} S_m.Rsd(hp_m)$. Por lo tanto, la operación de unión en combinación con la granularidad elegida, no es compatible con el invariante de consumo.

Ejemplo Consideremos el ejemplo polimórfico del método `map` presentado en el Código 6.1. La Figura 6.1 presenta el resultado del análisis de escape para la rama del grafo relacionado con el llamado polimórfico, en donde el nodo $I19$ proviene del método `Trans.apply` y los nodos $I13$, $I14$ e $I15$ de `Trans2.apply`.

Como se puede ver al tomar la unión de las implementaciones se generaron 2 subgrafos disjuntos accesibles a través del nodo $I2$. Tomando en cuenta que cada nodo es una partición residual, el residual total del método `map` se calcula como:

$$Rsd(newList) + \sum_{i=1}^{list.size} (\texttt{add}.Rsd(ret)(i) + \texttt{Trans.apply}.Rsd(ret)(i) +$$

$$\texttt{Trans2.apply}.Rsd(ret)(i) + \texttt{Trans2.apply}.Rsd(ret_0)(i) + \texttt{Trans2.apply}.Rsd(ret_1)(i)) =$$

$$= Rsd(I20) + \sum_{i=1}^{list.size} (Rsd(I2)(i) + Rsd(I19)(i) + Rsd(I13)(i) + Rsd(I14)(i) + Rsd(I15)(i)) =$$

$$= 1 + \sum_{i=1}^{list.size} (1 + 1 + 2 + 1 + 1) = 1 + 6 * list.size$$

Luego, la cantidad de memoria residual necesaria para ejecutar `map` de forma segura es $1 + 6 * list.size$. Este valor difiere del resultado obtenido $(1 + 5 * list.size)$ al realizar el análisis con una granularidad menor en el Capítulo 3. La diferencia se produce al tomar la unión de grafos como técnica para trabajar con el polimorfismo. Si bien esta operación resulta correcta para un análisis de escape, no lo es para el análisis de consumo ya que es necesario modelar que, si bien `Trans.apply` y `Trans2.apply` son polimórficos, no son ejecutados en simultaneo.

En las Secciones 6.4.1 y 6.4.2 se presentan dos alternativas para mejorar la precisión de las cotas obtenidas.

6.4.1. Coloreando nodos

El objetivo de esta alternativa consiste en modificar la forma en que se calcula el valor total del residual necesario para ejecutar de forma segura el método resumido.

El problema en el ejemplo anterior surge ya que a la hora de sumar los residuales aportados por cada partición, se suman valores de particiones provenientes de distintas implementaciones para una misma invocación virtual. A efectos de obtener una cota más baja es necesario considerar sólo la implementación que aporte mayor residual. Esto se logra agrupando las particiones pertenecientes a una misma implementación, acumulando el residual asociado a ellas, y luego tomando el máximo valor obtenido entre las diferentes implementaciones. Este proceso debe realizarse ante cada invocación realizada por el método resumido.

Notar que una vez agrupadas las particiones por implementación, para calcular el consumo residual de la misma puede ser necesario distinguir nuevamente entre invocaciones e implementaciones. Esto ocurre porque las particiones asociadas a una misma implementación pueden provenir de llamados polimórficos, es decir el método al cual pertenecen esas particiones puede también realizar llamadas polimórficas. Por lo tanto, para cada partición es necesario registrar los contextos de invocación.

Para esto se introduce la noción de Color y SubColor. Los mismos permiten discriminar invocaciones y, para las mismas, implementaciones. En caso que dos particiones provengan de una misma invocación, ignorando la implementación, diremos que tienen el mismo Color. En caso que dos particiones con el mismo Color provengan de una misma implementación diremos que tienen el mismo SubColor. Por lo tanto para mantener la información de los contextos de invocación, cada partición tendrá asociada una pila de tuplas $\langle \mathsf{Color}, \mathsf{SubColor} \rangle$, la cual será denominada pila de colores.

En el caso de la implementación de este trabajo se utiliza la pila asociada al análisis de escape desarrollado (ver Sección 5.5), la cual posee la suficiente información de contexto para representar la estructura requerida.

De esta forma, se define que una partición cuya pila de colores se encuentra vacía es una partición local para un método m, es decir, proviene de una instrucción `new` en el código de m. En este caso diremos que la partición no se encuentra coloreada. Por el contrario, una partición cuya pila de colores no se encuentra vacía, es una partición que fue generada por alguna de las invocaciones que m realiza. En este caso diremos que la partición se encuentra coloreada.

Se definen las siguientes funciones auxiliares para el algoritmo:

- $colorStack(hp)$: es una función que devuelve la pila de colores asociada a la partición hp.
- $popColorStack(hp)$: es una función que devuelve la partición hp' la cual se obtiene de remover el tope de la pila de colores asociada a hp.
- $getColors(\mathsf{HP})$: es una función que devuelve el conjunto de colores del tope de la pila asociada a cada partición del conjunto de particiones HP, es decir devuelve el conjunto $\{colorStack(hp).\mathsf{Color} \mid hp \in \mathsf{HP}\}$
- $getSubColors(\mathsf{HP})$: es una función que devuelve el conjunto de subcolores del tope de la pila asociada a cada partición del conjunto de particiones HP, es decir devuelve el conjunto $\{colorStack(hp).\mathsf{SubColor} \mid hp \in \mathsf{HP}\}$
- HP_{color}: es una expresión que representa las particiones del conjunto HP cuyo color del tope de la pila es $color$, es decir devuelve el conjunto $\{hp \mid hp \in \mathsf{HP}_{color} \ \wedge \ colorStack(hp).\mathsf{Color} = color\}$
- $\mathsf{HP}_{color,subColor}$: es una expresión que representa las particiones del conjunto HP cuyo color y subcolor del tope de la pila son $color$ y $subColor$ respectivamente, es decir devuelve el conjunto $\{hp \mid hp \in \mathsf{HP}_{color,subColor} \wedge colorStack(hp).\mathsf{Color} = color \wedge colorStack(hp).\mathsf{SubColor} = subColor\}$

El algoritmo 6.2 toma como entrada un resumen de consumo $\mathtt{S}_m$ y un conjunto de particiones residuales $resPart$, de forma tal que $resPart \subseteq Dom(\mathtt{S}_m.Rsd)$. Para obtener el valor residual total asociado a $\mathtt{S}_m.Rsd$ se lo debe invocar como $PartitionsToRsd(\mathtt{S}_m, Dom(\mathtt{S}_m.Rsd))$. Devuelve una expresión simbólica relacionada con el residual total del conjunto de entrada. Dado que en cada llamada el algoritmo sólo trabaja con el tope de la pila de cada partición, hablaremos de los colores y subcolores según los valores respectivos del tope de la pila asociada a cada una de ellas.

Para obtener el residual total, el algoritmo acumula el residual de las particiones no coloreadas (línea 4) lo que representa su caso base ya que el algoritmo se invoca recursivamente (línea 9).

Por otro lado toma las particiones coloreadas (línea 5) e itera por los diferentes colores existentes en las mismas, lo que representa iterar por los diferentes llamados

1: **procedure** PartitionsToRsd($\mathtt{S}_m$, $resPart$): $resTotal$
2: $resNews := resCalls := 0$
3: **for** each $hpNotColored \in \{hp \mid hp \in resPart \wedge\ colorStack(hp) = \emptyset\ \}$ **do**
4: $resNews := SA.add(resNews, \mathtt{S}_m.Rsd(hpNotColored))$
5: $hpColored := \{hp \mid hp \in resPart \wedge hp\ colorStack(hp) \neq \emptyset\ \}$
6: **for** each $color \in getColors(hpColored)$ **do**
7: $resCall := 0$
8: **for** each $subColor \in getSubColors(hpColored_{color})$ **do**
9: $resCall := SA.join(resCall, PartitionsToRsd(\mathtt{S}_m,$ $\{popColorStack(hp) \mid hp \in hpColored_{color,subColor}))\}$
10: $resCalls := SA.add(resCalls, resCall)$
11: $resTotal := SA.add(resNews, resCalls)$

Figura 6.2: Algoritmo Particiones a un único residual

(líneas 6-10). Luego toma los diferentes subcolores asociados a las particiones de cada color e itera sobre ellos (líneas 8-9), lo que representa iterar por el resumen de las diferentes implementaciones de un llamado. Luego obtiene el supremo de las expresiones simbólicas asociadas al residual total de las particiones asociadas al subcolor (línea 9), es decir, asociado a las implementaciones de un llamado. Esto se logra llamando recursivamente al algoritmo quitando el tope de la pila de colores a cada partición del conjunto (línea 9). Es necesario realizar un llamado recursivo pues la implementación a analizar recursivamente podría contener particiones provenientes de llamados polimórficos.

Luego se acumula el residual total de todos los colores (línea 10), es decir de todas las llamadas. Finalmente se acumula esta última expresión con la calculada para las particiones no coloreadas (línea 11). De esta forma, se obtiene una expresión simbólica relacionada al residual total de un conjunto de particiones.

Ejemplo La Figura 6.3 presenta el resultado del análisis de escape para el método `map` utilizando la estrategia desarrollada en esta sección. Se puede ver que los nodos $I13$, $I14$, $I15$ e $I19$ poseen el mismo Color en el relleno, lo que indica que provienen de una misma invocación (`Trans.apply`). Además los nodos $I13$, $I14$ e $I15$ poseen el mismo SubColor en el borde, lo que indica que provienen de una misma implementación (`Trans2`), mientras que el nodo $I19$ posee un Color distinto en el borde, es decir proviene de una implementación diferente (`Trans`). Notar que para poder graficar el ejemplo los nodos han sido coloreados con el tope de la pila. Si bien en este ejemplo las particiones tienen un único elemento en la pila asociada, las mismas

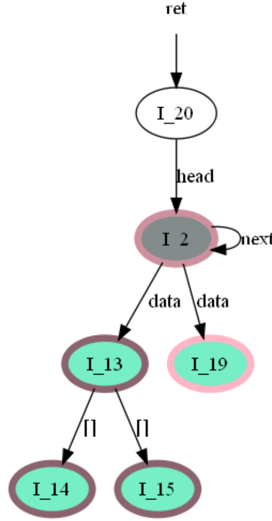


Figura 6.3: Resultado análisis escape para el método `map` con coloreo

podrían tener un tamaño mayor complejizando la representación.

El algoritmo comienza calculando el consumo proveniente de los nodos no coloreados, en este caso el consumo de $I20$. Luego itera por cada color. En el caso del color asociado al conjunto de nodos $\{I2\}$ realiza el llamado recursivo obteniendo que el residual de su color es el de su único nodo, es decir 1. Luego procede a analizar el color asociado a los nodos del conjunto $\{I13, I14, I15, I19\}$. Para esto divide los nodos en conjuntos según la implementación, es decir $\{I13, I14, I15\}$ y $\{I19\}$, y para cada conjunto realiza la llamada recursiva obteniendo como consumo 4 y 1 respectivamente, quedándose con el supremo de ambos.

El cálculo del residual total realizado por el algoritmo puede ser representado de la siguiente forma:

$$Rsd(newList) + \sum_{i=1}^{list.size} (\texttt{add}.Rsd(ret)(i) + max(\texttt{Trans.apply}.Rsd(ret)(i),$$

$$\texttt{Trans2.apply}.Rsd(ret)(i)+\texttt{Trans2.apply}.Rsd(ret_0)(i)+\texttt{Trans2.apply}.Rsd(ret_1)(i))) =$$

$$= Rsd(I20)+ \sum_{i=1}^{list.size} (Rsd(I2)(i)+max(Rsd(I19)(i), Rsd(I13)(i)+Rsd(I14)(i)+Rsd(I15)(i))) =$$

$$= 1 + \sum_{i=1}^{list.size} (1 + max(1, 2 + 1 + 1)) = 1 + \sum_{i=1}^{list.size} (1 + max(1, 4)) = 1 + 5 * list.size$$

De esta forma se puede ver que la estrategia obtuvo la expresión esperada para el residual total del método, mejorando la cota obtenida para la cantidad de memoria residual necesaria para ejecutar de forma segura un método.

6.4.2. Agrupando nodos

En la sección anterior se presentó una forma de abordar el problema generado por la presencia de llamados polimórficos basada en modificar el cálculo para obtener el valor total del residual. Otra posibilidad es trabajar sobre el Points-To graph obtenido del análisis de escape a fin de minimizar los efectos problemáticos causados por tomar la unión de grafos ante la aparición de llamadas polimórficas.

Esto se logra agrupando nodos de las distintas implementaciones siempre que puedan ser accedidos siguiendo el mismo camino en el grafo. Agrupar nodos produce un efecto de solapamiento de los grafos que participan de la unión. Esto busca producir un efecto que logre aproximar el resultado de la unión a un Points-To graph equivalente al máximo entre los argumentos. De esta forma se intenta restaurar el invariante de consumo invalidado al realizar la unión. Es importante mantener una equivalencia de caminos al agrupar puesto que el criterio de [SR05b, SR05a] para decidir si un nodo escapa o no se basa ellos. Entonces, mantener la equivalencia asegura que al agrupar el grafo no se está perdiendo información de escape.

En el ejemplo expuesto en la Figura 6.1 es posible juntar los nodos $I13$ e $I19$ tal como se observa en la Figura 6.4 puesto que son accedidos a través del campo *data*.

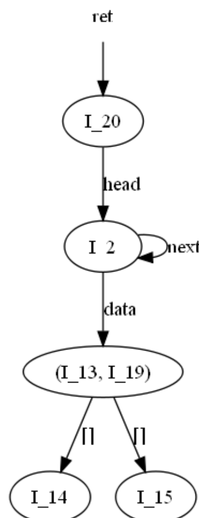


Figura 6.4: Resultado análisis escape para el método `map` con agrupando

Si se efectúa nuevamente el cálculo del total del residual es posible observar que para el ejemplo descripto se ha resuelto el problema pues el resultado es el esperado.

$$Rsd(newList) + \sum_{i=1}^{list.size} (\texttt{add}.Rsd(ret)(i) + max(\texttt{Trans.apply}.Rsd(ret)(i),$$

$$\texttt{Trans2.apply}.Rsd(ret)(i)) + \texttt{Trans2.apply}.Rsd(ret_0)(i) + \texttt{Trans2.apply}.Rsd(ret_1)(i)) =$$

$$= Rsd(I20) + \sum_{i=1}^{list.size} (Rsd(I2)(i) + max(Rsd(I19)(i), Rsd(I13)(i)) + Rsd(I14)(i) + Rsd(I15)(i)) =$$

$$= 1 + \sum_{i=1}^{list.size} (1 + max(1,2) + 1 + 1) = 1 + 5 * list.size$$

Esto ocurre porque al fusionar $I13$ e $I19$ en un sólo nodo, a la hora de vincularlos a través del EA serán enviados a la misma partición de memoria. De esta forma se toma el máximo del consumo aportado por cada nodo y no la suma.

Es interesante notar que esta estrategia minimiza los problemas pero todavía existen casos donde el problema puede persistir. Por ejemplo, si $I19$ representase un arreglo vacio de tamaño 10 nos encontraríamos nuevamente con que la cota obtenida por el algoritmo de consumo sería $1+13*list.size = 1+\sum_{i=1}^{list.size}(1+max(10,2)+1+1)$ cuando hubiese sido $1+11*list.size = 1+\sum_{i=1}^{list.size}(1+max(10,2+1+1))$ si se hubiese utilizado la técnica de coloreo antes mencionada.

El problema es que el criterio de agrupamiento elegido no fue capaz de solucionar el problema de la unión para todos los casos. Es interesante marcar esto pues es posible variar los criterios de agrupamiento para lograr obtener grafos donde el problema sea resuelto. El problema con estos criterios es que si agrupan demasiados nodos puede perderse precisión en lo que al análisis de escape se refiere. Por lo tanto, la estrategia a elegir debe asumir un compromiso entre precisión de escape y precisión en la cota obtenida.

En conclusión, otra forma de abordar el problema que presentan las llamadas polimórficas en la instanciación del algoritmo de consumo propuesta es agrupar bajo algún criterio los nodos del Points-To graph obtenidos por el análisis de escape. Esta solución puede no resolver todos los escenarios problemáticos pero tiene la ventaja que no necesita modificar la representación del resumen de memoria. Es interesante notar que el hecho de reducir la cantidad de nodos no sólo aporta a la solución de la problemática de las llamadas polimórficas sino que impacta en la performance del algoritmo de consumo ya que disminuye la cantidad de particiones de memoria con las que se trabaja.

6.5. Limitaciones

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

Por otro lado debido al SA implementado, 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 se podría continuar manipulando estas expresiones.

Determinar el máximo entre un conjunto de polinomios de manera simbólica es un problema que no puede ser resuelto completamente sin que la cantidad de candidatos en el resultado crezca de forma importante y, por lo tanto, crezca también el costo de calcularlo. Esto se debe a que 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. A efectos de este trabajo es suficiente con la solución propuesta.

La alternativa propuesta para *colorear* nodos en 6.4.1 resuelve de forma precisa los llamados polimórficos. Sin embargo, la forma en que resuelve el problema agrega una importante carga de operaciones al algoritmo, lo que trae como consecuencia un importante deterioro en la performance a la hora de presentar los resultados del mismo, momento en el que se calcula el valor total del residual. La alternativa propuesta para *agrupar* nodos en 6.4.2 no resuelve el problema con la misma precisión.

6.6. Conclusiones

En este capítulo presentamos una instanciación del algoritmo descripto en el Capítulo 4, definida por la implementación de las herramientas auxiliares necesarias para el algoritmo (CT, SA y EA), y por la representación de las expresiones paramétricas y las particiones de memoria.

Utilizamos una serie de operadores de la calculadora `iscc` para dar soporte al CT y al SA, presentamos limitaciones de los mismos y decidimos utilizar la estrategia *lazy* junto con el operador "." en busca de la mejor precisión posible.

Definimos el concepto de partición de memoria basándonos en el análisis de [SR05b, SR05a] y como dar soporte a los operadores del EA.

Presentamos la problemática que surge a partir de la granularidad del análisis de escape utilizado para los casos en que existan llamadas polimórficas y dos estrategias

para combatirla. Esto abre un posible trabajo a futuro el cual consiste en mejorar la performance de la primer alternativa, la precisión de la segunda de ellas, o bien modificar la forma en que se aborda el problema de las llamadas polimórficas (Ver Sección 9.3).

En el capítulo siguiente se presentan los aspectos técnicos relevantes sobre la herramienta implementada para evaluar la instanciación del algoritmo descripta en este capítulo.

Capítulo 7

Implementación

En el capítulo anterior presentamos una instanciación del algoritmo descripto en el Capítulo 4, definida por la implementación de las herramientas auxiliares necesarias para el algoritmo (CT, SA y EA), la representación de las expresiones paramétricas y las particiones de memoria, y una serie de alternativas para mejorar las cotas obtenidas en los casos en que se presenten llamadas polimórficas.

En este capítulo se presentan los aspectos técnicos relevantes sobre la herramienta implementada, con el objetivo de evaluar la instanciación del algoritmo descripta. Como se describió en secciones anteriores (ver Figura 1.1) la solución está compuesta por cuatro grandes módulos:

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

7.1. Proveedor de invariantes

Este componente es el encargado de brindar los invariantes para los puntos relevantes del programa requeridos por la implementación del CT y del SA. Los invariantes pueden ser especificados usando la API Java que provee la herramienta o mediante un archivo con formato XML (Ver apéndice A.1).

Si bien el *proveedor de invariantes* es un componente que implementa una función específica, desacoplarlo del resto de la herramienta permite realizar el análisis de consumo de forma independiente del problema del cálculo de invariantes. De esta forma, es posible delegar la inferencia y especificación de los mismos a alguna herramienta capaz de producir una salida en el formato XML esperado por nuestro componente [Gar05, ECGN99].

Si bien no es el objetivo de esta tesis trabajar sobre el problema de la generación de invariantes, contar con invariantes precisos constituye una parte importante de la instanciación presentada pues afecta directamente a la precisión de la misma. Un posible trabajo a futuro consiste en realizar la integración con alguna herramienta que permita extraer invariantes especificados en el código usando algún lenguaje de especificación como, por ejemplo, JML [LLP+00]. Esto, en conjunto con la integración de herramientas que generen invariantes de forma automática como la presentada en [Gar07], aumentaría la precisión y usabilidad de la herramienta. Además facilitaría el análisis de consumo de programas complejos.

7.2. Analizador de información de escape

Este componente implementa las operaciones requeridas por el EA, en otras palabras, determina la partición de memoria asociada a un objeto creado localmente y establece la relación entre las particiones del método cliente y las de aquellos objetos que exceden el alcance del método invocado.

La existencia de este componente permite reemplazar la implementación del *Análisis de escape* por cualquier otra que respete las operaciones definidas por el EA.

El *Análisis de escape* implementado es el presentado en la Sección 5 el cual está basado en [SR05b, SR05a]. Existe una implementación de este trabajo la cual forma parte del framework para optimización de programas Java llamado Soot [VRHS+99]. Dentro del framework es posible encontrarla bajo el nombre de *Purity Analysis* y fue utilizada como base para el desarrollo del componente.

El *Análisis de escape* puede ejecutarse de forma independiente al *Análisis de memoria dinámica*. El resultado del mismo son resúmenes de escape en formato XML que son utilizados durante el análisis de consumo. Esto permite reutilizar los resúmenes de escape ya calculados para distintas ejecuciones del análisis de consumo. Esta característica aporta escalabilidad al prototipo desarrollado. En el apéndice A.2 se explica la forma de ejecutar el componente. El formato del XML es particular a la implementación del análisis de escape realizada. No es más que la serialización de los resúmenes obtenidos, razón por la cual no es detallado.

Durante la experimentación realizada se detectó que un programa de tamaño medio como *JLayer* (ver Sección 8.3), utiliza una gran cantidad de variables estáticas para enviar parámetros entre métodos de forma eficiente. Si bien son objetos estáticos, por las características del programa se observa que pueden ser recolectados al finalizar la ejecución de ciertos métodos. A pesar de que este aspecto no puede ser detectado por los análisis de escape, es deseable poder especificarlo pues permite modelar la naturaleza de los programas de forma más precisa. Por lo tanto, se introdujo una variante que permite especificar cuándo los objetos que fueron considerados

residuales por el análisis deben ser capturados.

7.3. Calculadora de expresiones paramétricas

Este componente es el encargado de proveer un entorno para operar con expresiones paramétricas. En conjunto con el componente de *Invariantes Locales* permite implementar las herramientas CT y SA necesarias para el análisis de consumo de memoria.

Para implementar los distintos operadores de la calculadora se desarrollo un conjunto de clases Java que permiten interactuar vía línea de comandos con la calculadora `iscc` , la cual forma parte de la librería [Ver07, Ver10]. Utilizar esta vía de comunicación genera un overhead en la performance causado por la generación del string necesario para realizar la invocación a la línea de comandos y por el parseo del resultado para su posterior procesamiento. Este problema puede ser abordado, por ejemplo, minimizando los llamados a la calculadora `iscc` diagramando una estrategia de consultas a fin de minimizar las invocaciones a la misma

Se definió una API para la calculadora a fin de poder reemplazar la implementación propuesta si el overhead resultase inaceptable. La API permite, por ejemplo, implementar otra versión de la calculadora basándose en *Jbarvinok* como se hizo en [RG09].

A pesar del overhead se decidió utilizar `iscc` porque permite trabajar con las operaciones definidas en [Ver07, Ver10] sin tener que hacer uso de alguna herramienta para integrar C/C++ a Java. Además, la interfaz en modo texto permite tener una abstracción de las estructuras de datos utilizadas por la librería (e.g. representación de polítopos, dominios, etc). De esta forma, si surge una nueva versión de la librería, se podrá utilizarla siempre y cuando la calculadora mantenga la misma sintaxis.

La calculadora trabaja con polinomios restringidos a dominios, lo que concuerda con la definición de expresiones paramétricas utilizadas. Los operadores utilizados para el desarrollo de este componente fueron "*card*", "*sum*", "*ub*", "." y "+".

El conjunto de clases Java desarrolladas combina estos operadores a fin de implementar las operaciones *count*, *summate*, *maximize*, *join* y *add* requeridas por el CT y el SA.

Las operaciones que representaron un mayor desafío fueron las de *maximize* y *join*. Como se explicó en la Sección 6.3.2, la librería no logra obtener siempre un único polinomio para el valor máximo (en algunos casos es un problema complejo y muy difícil de resolver) pudiendo retornar un conjunto de candidatos. Trabajar con candidatos en lugar de con expresiones concretas impacta negativamente en la performance de todos los operadores.

Como se menciono anteriormente se definen dos estrategias de comparación: *lazy* y *add*. El componente implementado permite ejecutar el análisis utilizando cualquiera

de las dos. Esto permite variar entre precisión y performance para las distintas ejecuciones del análisis.

Por último, en la implementación realizada no se ha hecho uso de todo el poder expresivo de la librería [Ver07, Ver10]. La misma soporta restricciones existenciales (*exists*) las cuales no son soportadas por el componente. En otras palabras, el componente no soporta que la `iscc` devuelva como resultado un dominio con restricciones existenciales ni soporta como entrada dominios con estas características. Permitir el uso de los mismos constituye un posible trabajo a futuro pues esto aumentaría el poder expresivo del que se dispone para predicar sobre los dominios a los cuales están restringidas las expresiones paramétricas.

7.4. Analizador de memoria dinámica

Este componente implementa el algoritmo composicional para inferencia de requerimientos de memoria. El componente fue construido sobre el framework para optimización de programas Java llamado Soot [VRHS+99]. Este framework permite generar *call graphs* e implementar varios análisis sobre el flujo de datos. Esto permitió entre otras cosas implementar el componente de *call graph*.

La inferencia de resúmenes es implementada mediante el algoritmo presentado en la Sección 4.1. Para esto se utiliza las funcionalidades de Soot para generar el *call graph* y procesar cada uno de los métodos del programa. El componente se apoya en la implementación del CT, SA y EA para llevar a cabo todas las operaciones. Los primeros dos fueron implementados combinando los componentes de *Invariantes Locales* y *Calculadora de Expresiones Paramétricas* mientras que las operaciones del último son resueltas por el componente de *Información Escape*. Como salida del análisis se genera un resumen de consumo en formato XML y un reporte el cual presenta la especificación obtenida para cada uno de los métodos analizados.

Al igual que el componente de escape, este componente tiene la capacidad de reutilizar los resúmenes generados. Esto significa que no se analizarán métodos para los que se posea un resumen, aportando escalabilidad a la herramienta.

Si bien existe una interfaz que debe respetar todo resumen de memoria (ver 2.3) la estructura con la cual se almacenan puede variar dependiendo del análisis de escape utilizado. Por ejemplo para la instanciación realizada, al basarse en [SR05b, SR05a], el resumen es un Points-To graph donde cada nodo está decorado con información de consumo. Esto hace que la noción de partición de memoria pueda cambiar dependiendo del análisis auxiliar de escape utilizado, lo que dificulta usar los resúmenes de una implementación en otra. La tarea de generar un lenguaje de especificación general constituye un interesante trabajo a futuro. Esto permitiría escribir resúmenes en un formato estándar y facilitaría la interacción entre distintas instanciaciones del algoritmo, así como también con otras herramientas capaces de trabajar con estos

resúmenes. Por ejemplo, podría ser la entrada para un verificador capaz de determinar si la información inferida es correcta.

Una forma de mejorar la performance de la implementación desarrollada para este componente es agregar al resumen de un método m el valor total del residual a fin de reutilizarlo en el cálculo del residual de los métodos que invocan a m. De esta forma se podría reemplazar la llamada recursiva presente en el algoritmo 6.2 por el valor total del residual previamente calculado.

En el apéndice A.3 se explica cómo ejecutar el componente.

Capítulo 8

Experimentos y resultados

Para evaluar tanto la técnica como la herramienta desarrollada en esta tesis, se realizaron experimentos sobre los ejemplos presentados en la Sección 3, programas del *benchmark* JOlden [CM01] y sobre *JLayer*, una librería JAVA de código abierto que convierte, reproduce y decodifica MPEG.

La expresión $Tmp = S_m.Tmp$ se refiere al consumo de memoria temporal estimado de un método m, mientras que $Rsd = \sum_{hp_m \in Dom(S_m.Rsd)} S_m.Rsd(hp_m)$ al consumo residual total. Además, se utiliza el estimador paramétrico $MemReq_m = S_m.Tmp + \sum_{hp_m \in Dom(S_m.Rsd)} S_m.Rsd(hp_m)$ (Ver Capítulo 2), que aproxima el mayor número de objetos vivos reservados durante la ejecución de m y los métodos invocados por m. El objetivo del mismo es contrastar la cantidad de objetos estimados para un método m contra el consumo real de m.

El objetivo de los primeros experimentos fue evaluar la precisión de la herramienta desarrollada y compararla con otras técnicas, mientras que el objetivo del experimento sobre *JLayer* fue evaluar la analizabilidad (i.e. el comportamiento del análisis en programas de mayor tamaño). Para comparar los resultados obtenidos por el análisis composicional se utilizó la herramienta desarrollada en la tesis de licenciatura de Rouaux [RG09]. Además para los experimentos de JOlden se utiliza el valor de $MemReq$ obtenido según [BFGY08]. Este trabajo sólo se utiliza para este experimento sin presentar los tiempos totales ya que no pudo ser ejecutada, por lo que se toman los resultados del informe presentado.

En el Cuadro 8.1 se presenta un resumen de los experimentos realizados, incluyendo los tiempos (seg) de la herramienta desarrollada para las alternativas de agrupar y colorear nodos, y la presentada en [RG09] (Virtual Machine, Core 2 duo 1.6Ghz, 2GB RAM). Los mismos se detallan según el tiempo de análisis de escape (Esc), análisis de consumo (Con), presentación de resultados (Pre) y el tiempo total (Tot). La importancia de los tiempos de la presentación de resultados se debe a que la estrategia de colorear nodos impacta directamente en esta etapa, ya que es una forma de calcular el resultado total que se va a presentar. Para la herramienta de

[RG09] sólo se presentan los tiempos totales.

Se puede ver que los tiempos de la herramienta desarrollada para ambas estrategias son en general algo mayores, pero aceptables, a los de la presentada en [RG09]. Sin embargo, en el caso de *JLayer* los tiempos son varias veces mayores. Esto resulta lógico ya que se buscó mejorar la precisión del análisis con el menor sacrificio de performance posible, algo que se logro llevar a cabo sin mayores problemas en los casos no polimórficos, pero que tuvo sus importantes consecuencias en los experimentos polimórficos debido a la cantidad de operaciones a realizar por las características del análisis y la herramienta realizadas.

Si bien se presentó en el Capítulo 6.4.2 que la estrategia que agrupa nodos no era tan precisa como la que colorea nodos, en los experimentos realizados resultó ser más performante sin resignar *nunca* precisión. Esta mejora en la performance se debe a que al obtener un grafo con una menor cantidad de nodos, el análisis trabaja con menos particiones y por ende realiza una menor cantidad de operaciones. De aquí en adelante se presentan sólo los resultados de la estrategia de *agrupar* nodos.

Experimento	♯Clases	♯Métodos	♯Líneas	*Agrupar* (seg)				*Colorear* (seg)				[RG09](*seg*)
				Esc	*Con*	*Pre*	*Tot*	*Esc*	*Con*	*Pre*	*Tot*	*Tot*
Ej.Motiv	6	22	84	3	17	2	22	4	18	3	25	20
MST	6	34	375	5	25	3	33	6	28	5	39	24
EM3D	4	21	233	5	27	3	35	7	31	6	44	18
JLayer	36	229	12544	80	334	47	461	532	1734	252	2518	131

Cuadro 8.1: Resumen experimentos realizados.

8.1. Ejemplos motivacionales

Los primeros ejemplos que se van a considerar son los presentados en el Capítulo 3. Estos ejemplos resultan interesantes ya que permiten ver el comportamiento de la técnica de análisis de consumo presentada, tanto para el algoritmo como para su instanciación. Además, se puede observar la manipulación simbólica de polinomios y la efectividad de las operaciones simbólicas desarrolladas.

El cuadro 8.2 presenta los resultados obtenidos por la herramienta desarrollada y por la herramienta desarrollada en [RG09] para los distintos métodos que constituyen el programa. Se puede ver que para la herramienta desarrollada en este trabajo los resultados son los estimados en el Capítulo 3 y reflejan de manera exacta el residual y temporal de los distintos métodos. Sin embargo para la técnica presentada en [RG09] se obtienen resultados con una menor precisión. Esto se debe principalmente a la captura del iterador por parte de la herramienta desarrollada en este trabajo. Esta mejora en la precisión se refleja no sólo en el método en donde el iterador es capturado, sino también en los métodos llamadores. De esta forma se puede apreciar con ejemplos simples cómo se pueden lograr los objetivos buscados, es decir la mejora en la precisión del análisis.

Método	Tmp	Rsd	$Tmp_{[RG09]}$	$Rsd_{[RG09]}$
multiply	$v.length$	$2 * v.length$	0	$3 * v.length$
testMultiply	$2 * v1.length + 2 * v2.length + max\{v1.length, v2.length\}$	0	0	$6 * v.length$
Trans.apply	2	1	2	1
Trans2.apply	1	4	1	4
map	3	$5 * list.size + 1$	2	$5 * list.size + 2$
copy	1	$list.size + 1$	0	$list.size + 2$
safemap	$list.size + 4$	$5 * list.size + 1$	1	$6 * list.size + 4$
test	$maxSize + 5$	$((5 * maxSize + 1) * list.size) + list.size + 1$	1	$(6 * maxSize + 4) * list.size) + list.size + 2$

Cuadro 8.2: Resumen de consumo por método para los ejemplos motivacionales.

8.2. JOlden

Los programas MST y EM3D del *benchmark* JOlden también fueron evaluados. Para estos programas se calcularon los resúmenes por método utilizando la técnica presentada. La estimación de consumo del método principal de ambos programas es comparada con los valores obtenidos en la herramienta desarrollada en [RG09] que utiliza un análisis de escape basado en regiones. Además se compara el valor de $MemReq$ con el obtenido en [BFGY08].

8.2.1. 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. El Cuadro 8.3 presenta las estimaciones de consumo para los métodos relevantes de la aplicación obtenidas por la técnica presentada y por la herramienta desarrollada en [RG09].

Método	Tmp	Rsd	$Tmp_{[RG09]}$	$Rsd_{[RG09]}$
doAllBlueRule	0	1	0	1
computeMST	$numvert$	0	$numvert$	0
addEdges	0	$2 * numvert^2$	0	$2 * numvert^2$
parseCmdLine	2	1	3	0
mainParameters	$6 + 2 * numvert$	$2 * numvert + 3 * numvert^2$	6	$4 * numvert + 3 * numvert^2 + 1$

Cuadro 8.3: Resumen de consumo por método para MST.

Se puede ver en el cuadro 8.3 que existe una diferencia en el método `parseCmdLine` en el cual la herramienta de [RG09] captura un objeto más. Esto se debe a que en esta herramienta se estaba dando un tratamiento impreciso a las excepciones, algo que fue modificado en este trabajo pudiendo capturar algunas excepciones lanzadas por el programador. Es decir que si bien en este trabajo se capturan menos objetos para este método, el resultado es de mayor precisión. Además, la instanciación presentada captura una serie de objetos en `mainParameters` que no son capturados en [RG09]. Esta resulta ser una importante mejora ya que se relaciona con uno de los parámetros del método, lo que implica que se capturen una cantidad paramétrica mayor de objetos. Sin embargo el valor de $MemReq$ coincide en ambas herramientas, la diferencia radica en si los objetos son o no capturados por algunos de los métodos invocados.

El Cuadro 8.4 presenta el valor estimado de consumo por la herramienta desarrollada, el valor estimado según [RG09], el valor estimado según [BFGY08] y el consumo real del método $mainParameters$ en función del parámetro $numvert$. Además, presenta el error de las herramientas respecto del consumo real. Como se mencionó anteriormente en este ejemplo el valor estimado para $MemReq$ es igual

para la estrategia de agrupar nodos y para [RG09]. Si bien a medida que aumenta la cantidad de objetos el error disminuye, la diferencia entre el valor estimado y el real es bastante amplia. Esto se debe a que al no poder ingresar restricciones de tipo *exists* no se cuenta con el poder suficiente para proveer los invariantes que acoten los objetos de una mejor manera y por lo tanto el resultado se ve sobreaproximado (Ver Sección 9.1). En este ejemplo particular esta limitación imposibilita generar la restricción $1/4 * numVert$ en uno de los métodos llamados, con la cual se hubiese obtenido el resultado exacto. Para el caso de [BFGY08] la estimación resultó ser más aproximada ya que por la forma con la que se trabaja con la librería de Barvinok no se cuenta con este problema.

numvert	$MemReq$ estimado			$MemReq$	$error$		
	$Agrupar$	[RG09]	[BFGY08]		$Agrupar$	[RG09]	[BFGY08]
50	7707	7707	5829	5700	47, 49 %	47, 49 %	2, 26 %
75	17182	17182	12960	12750	34, 76 %	34, 76 %	1, 65 %
100	30407	30407	22904	22700	33, 95 %	33, 95 %	0, 89 %

Cuadro 8.4: $MemReq$ y error para el consumo de MST.

8.2.2. EM3D

Al igual que la aplicación MST, el programa EM3D construye un grafo G en función del parámetro nN (número de nodos), donde cada nodo tendrá grado nD (número de grado). Luego, ejecuta nI (número de iteraciones) veces un algoritmo sobre G. El Cuadro 8.5 presenta las estimaciones de consumo para los métodos relevantes de la aplicación EM3D obtenidas por la técnica desarrollada y por la herramienta desarrollada en [RG09].

Método	Tmp	Rsd	$Tmp_{[RG09]}$	$Rsd_{[RG09]}$
create	$6 + 2 * nN$	$2 + 2 * nN + 6 * nN * nD$	1	$8 + 4 * nN + 6 * nN * nD$
parseCmdLine	3	3	6	0
compute	2	0	0	2
mainParameters	$11 + 4 * nN + 6 * nD * nN$	1	$12 + 4 * nN + 6 * nD * nN + 2 * nI$	1

Cuadro 8.5: Resumen de consumo por método para EM3D.

Se pueden ver interesantes aspectos en la comparación de los resultados presentados por las diferentes herramientas. El primer punto a analizar son los 2 objetos capturados en `compute`. Más allá de ser un valor constante, lo interesante de este número es que está relacionado con la captura de iteradores, patrón que no era bien modelado en [RG09]. Luego, el hecho de capturar estos objetos hace una importante mejora en `mainParameters` al eliminar del resultado al parámetro nI, ya que el método `compute` se ejecuta nI veces.

El Cuadro 8.6 presenta el valor estimado de consumo por la herramienta desarrollada, el valor estimado según [RG09], el valor estimado según [BFGY08] y el consumo real del método *mainParameters* en función de los parámetros nN y nI

para un valor fijo de nD igual a 1. Además, presenta el error de las herramientas respecto del consumo real. En este caso, el valor obtenido para $MemReq$ por la herramienta desarrollada en este trabajo obtiene una estimación exacta de los objetos reales. Además este valor es similar al presentado en [BFGY08]. Por otro lado el valor estimado para $MemReq$ difiere con el obtenido en [RG09], principalmente debido al parámetro nI y a cómo se manipulen sus valores. Esto puede traer como consecuencia un alto porcentaje de error en la herramienta de [RG09], lo que muestra lo importante de mejorar la precisión del análisis para patrones que no eran bien modelados. La Figura 8.1 refleja este comportamiento de manera gráfica.

nN	nI	$MemReq$ estimado			$MemReq$	$error$		
		$Agrupar$	[RG09]	[BFGY08]		$Agrupar$	[RG09]	[BFGY08]
20	1	212	215	214	212	0,00 %	1,41 %	0,94 %
20	10	212	233	214	212	0,00 %	9,90 %	0,94 %
20	50	212	313	214	212	0,00 %	47,64 %	0,94 %
50	1	512	515	514	512	0,00 %	0,58 %	0,39 %
50	10	512	533	514	512	0,00 %	4,10 %	0,39 %
50	50	512	613	514	512	0,00 %	19,73 %	0,39 %
100	1	1012	1015	1014	1012	0,00 %	0,30 %	0,20 %
100	10	1012	1033	1014	1012	0,00 %	2,07 %	0,20 %
100	50	1012	1113	1014	1012	0,00 %	9,98 %	0,20 %

Cuadro 8.6: $MemReq$ y error para el consumo de EM3D.

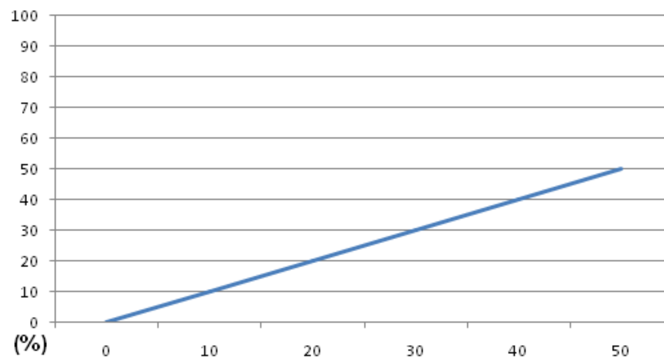


Figura 8.1: Crecimiento %error en [RG09] al aumentar el valor de nI

8.3. JLayer

Esta aplicación constituye el caso de estudio más importante utilizado para evaluar la técnica desarrollada. *JLayer* es una librería de código abierto para JAVA que decodifica, reproduce y convierte MPEG 1/2/2.5 de 1/2/3 capas. Este *benchmark* es interesante ya que posee una versión para J2ME que corre en dispositivos compatibles con CLDC (como móviles) en donde la memoria es un recurso limitado. Además *JLayer* es un programa de tamaño medio que involucra 36 clases y 12544 líneas de código.

El Cuadro 8.7 presenta las estimaciones de consumo para los métodos relevantes de la aplicación *JLayer* obtenidas por la técnica desarrollada. Los parámetros con los que trata la aplicación son fL (número de capa), fi (intensidad del stereo), fs (número de subbandas), m (modo), di (decodificador inicializado) y mS (máximo de subbandas). Además se introduce una variable mB para mejorar la precisión del análisis buscando proveer el tamaño del buffer más grande que se va a escribir por el programa bajo análisis. Es responsabilidad de quien analiza el programa proveer el valor de esta variable.

<table>
<tr><th>Método</th><th>Tmp</th><th>Rsd</th></tr>
<tr><td>Decoder.
retrieveDecoder
(layer)</td><td>0</td><td>$L = 1 : 2$
$L = 2 : 2$
$L = 3 : 41076$</td></tr>
<tr><td>Decoder.
decodeFrame
(frame f)</td><td>1</td><td>$fL = 1 : fs + 34$
$fL = 2 \ \&\& \ m = 1 : 10fs + 7fi + 34$
$fL = 2 \ \&\& \ m = 2 : 17fs + 34$
$fL = 2 \ \&\& \ m = 3 : 10fs + 34$
$fL = 3 : 0$</td></tr>
<tr><td>Player.
decodeFrame
(frame f)</td><td>$fL = 1 : 2 * mB + fs + 1094$
$fL = 2 \ \&\& \ m = 1 : 2 * mB + 10fs + 7fi + 1094$
$fL = 2 \ \&\& \ m = 2 : 2 * mB + 17fs + 1094$
$fL = 2 \ \&\& \ m = 3 : 10fs + 5700$
$fL = 3 : 2 * mB + 1060$</td><td>$fL = (1||2) \ \&\& \ m = (1||2) \ \&\& \ !di : 5662$
$fL = (1||2) \ \&\& \ m = 3 \ \&\& \ !di : 3988$
$fL = 3 \ \&\& \ m = (1||2) \ \&\& \ !di : 46736$
$fL = 3 \ \&\& \ m = 3 \ \&\& \ !di : 45062$</td></tr>
<tr><td>jlp.Play</td><td>$fL = 1 : 2 * mB + mS + 10241$
$fL = 2 \ \&\& \ m = 1 : 2 * mB + 17 * mS + 10234$
$fL = 2 \ \&\& \ m = 2 : 2 * mB + 17 * mS + 10241$
$fL = 2 \ \&\& \ m = 3 : 2 * mB + 10 * mS + 9121$
$fL = 3 : 2 * mB + 49486$</td><td>$fL = (1||2) \ \&\& \ m = (1||2) \ \&\& \ !di : 2855$
$fL = (1||2) \ \&\& \ m = 3 \ \&\& \ !di : 2301$
$fL = 3 \ \&\& \ m = (1||2) \ \&\& \ !di : 3530$
$fL = 3 \ \&\& \ m = 3 \ \&\& \ !di : 2976$</td></tr>
</table>

Cuadro 8.7: Resumen de consumo por método para JLayer.

Se puede ver que la herramienta desarrollada obtiene una gran cantidad de casos entre los resultados. Esto se debe a que por las características del programa analizado se genera diferente consumo según el valor de fL y el resto de los parámetros. Sin embargo lo interesante del análisis es el hecho que la herramienta desarrollada permite analizar programas reales como ser un reproductor de $MPEG$.

El Cuadro 8.8 presenta el valor estimado de consumo por la herramienta desarrollada, el error respecto del consumo real y el consumo real de la aplicación en función de los parámetros fL, m y ms para un valor fijo de $mB = 2306$ y $di = 0$.

En este caso el valor estimado para $MemReq$ obtenido por la herramienta, difiere del valor real en un porcentaje el cual se considera aceptable para el tamaño del programa. Si bien este porcentaje varía según los parámetros, esta diferencia resulta

fL	m	ms	$MemReq$ estimado	$MemReq$	$error$
2	3	12	17909	17061	4,97 %
2	3	27	10701	10609	0,86 %
3	3	27	64509	64454	0,08 %
3	1	30	71784	71429	0,49 %

Cuadro 8.8: $MemReq$ y error para el consumo de JLayer.

mínima respecto al total de objetos.

La diferencia obtenida se debe principalmente a que para simplificar las expresiones de consumo obtenidas, muchas de las instrucciones de tipo condicional no fueron modeladas. Esto significa que para las instrucciones que forman parte del condicional, en sus invariantes, no se agrega una condición que exprese que la cantidad de visitas a la misma depende del resultado del condicional.

Por ejemplo, sea el código de programa `if a == null {instr1} else {instr2}`. Para modelar de forma correcta el condicional, el invariante asociado a `instr1` debe contener una condición del estilo de $isNull(a) == 1$, mientras que el de la instrucción `instr2` del estilo de $isNull(a) == 0$. Cuando se dice que no se modela el condicional es porque no se incluye ninguna de estas condiciones en dichos invariantes.

En los casos donde la ganancia de precisión no era suficiente, se decidió simplificar el experimento evitando generar nuevos parámetros relevantes y sobreaproximando el consumo. En otras palabras, no se modelaron condicionales cuando el beneficio de hacerlo no justificaba el trabajo necesario para hacerlo.

Por ejemplo las condiciones de control de errores no se modelaron con el objetivo de simplificar el análisis. Si bien esto genera una sobreaproximación ya que en la práctica son pocas las veces que las guardas de control resultan verdaderas, es decir, son pocas las veces que se consume lo que está en las guardas, modelarlas implica realizar un análisis más complejo que no resulta en una mejora importante de la precisión.

En conclusión el resultado obtenido es razonable para el tamaño del programa analizado dado la complejidad del mismo, la cantidad de parámetros de los que depende y los motivos que generaron el desvió en el resultado. Resulta interesante realizar nuevas experimentaciones con programas de similar tamaño a fin de evaluar optimizaciones generales para las cotas obtenidas como la presentada en la Sección 7.2.

Capítulo 9

Conclusiones

Hemos presentado un algoritmo de análisis composicional que soporta instanciaciones de diferentes técnicas capaces de aproximar el tiempo de vida de un objeto y aborda el problema de polimorfismo con el objetivo de mejorar las cotas obtenidas. El algoritmo brinda la posibilidad de desarrollar distintas instanciaciones del mismo cambiando el análisis de escape utilizado y/o variando la representación utilizada para las expresiones paramétricas.

Además brindamos una instanciación del algoritmo que utiliza un análisis de escape basado en [SR05b, SR05a], adaptado con la posibilidad de ser k-sensitivo, mostramos diferentes estrategias para resolver los operadores necesarios, presentamos los problemas que trae como consecuencia utilizar la granularidad elegida para los casos polimórficos y propusimos diferentes alternativas para abordarlos.

Por otro lado desarrollamos un mecanismo de inferencia que aprovecha las características de las nuevas versiones de la librería Barvinok [Ver07, Ver10], maximizando el uso de las operaciones que la misma provee y mejorando los resultados obtenidos, brindando la posibilidad de postergar el momento en que se da el resultado concreto de una operación hasta el momento en que se cuenta con la información necesaria para que éste sea lo más exacto posible.

En comparación con la herramienta desarrollada en [RG09] podemos decir que la implementación presentada mejora la precisión para algunos patrones clásicos como los iteradores y obtiene resultados similares en los casos polimórficos. Sin embargo la precisión ganada tuvo un costo importante en la performance. Por otro lado la estrategia desarrollada para agrupar nodos resultó en los experimentos realizados tan precisa como la que colorea nodos siendo a la vez más performante. Finalmente, es importante destacar que el enfoque y la herramienta implementada fueron evaluados con un programa JAVA de tamaño medio, obteniendo resultados satisfactorios.

9.1. Limitaciones

En la Sección 5.6 mostramos que el análisis no ofrece la posibilidad de detectar cuándo un objeto que escapa globalmente puede considerarse capturado. Además, la asignación a campos de un objeto siempre produce un weak update. Por otro lado, se utiliza una única etiqueta para relacionar arreglos con su contenido. Si bien estas características favorecen a que el análisis sea simple, como consecuencia se pierde precisión en cuanto a los objetos que efectivamente son capturados.

En la Sección 6.5 mostramos que las alternativas propuestas para abordar el problema de llamadas polimórficas obligan a elegir entre performance y precisión. Por otro lado, tanto la operación de suma como la de máximo de una expresión paramétrica sobre un dominio se restringen a expresiones polinomiales, mientras que determinar el máximo entre un conjunto de polinomios de manera simbólica es un problema que no puede ser resuelto completamente.

Más allá que el enfoque presentado permite tratar con programas de mayor tamaño, aún en presencia de polimorfismo, algunos patrones como el de recursión no son tratados. Además, los invariantes utilizados continúan siendo lineales y numéricos tal como ocurría en [RG09], lo que limita el poder expresivo. Por otro lado la utilización de invariantes locales, si bien son sencillos y manipulables, proporcionan menor información de contexto que los invariantes globales. Ambos factores representan para algunos programas la fuente de imprecisión en su análisis de consumo.

Por último, en la implementación realizada no se ha hecho uso de todo el poder expresivo de la librería [Ver07, Ver10]. La librería soporta restricciones existenciales (*exists*) las cuales no son soportadas por el componente. Permitir el uso de las mismas aumentaría el poder expresivo del que se dispone para predicar sobre los dominios a los cuales están restringidas las expresiones paramétricas.

9.2. Trabajos relacionados

Durante los primeros años de la última década la mayoría de los trabajos relacionados estaban 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].

Sin embargo, en los últimos años creció considerablemente la cantidad de trabajos relacionados con la inferencia de requerimientos de memoria dinámica [Ghe02, HJ03, CJPS05, USL03, AGGZ10, BGY06, GZ10].

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 última 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 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 está 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, mientras que las expresiones que soporta se limitan a la aritmética 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 *garbage collection* para lenguajes funcionales. 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 [BGY06, BFGY08] se plantea una técnica 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 éste 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. 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. Estas técnicas requieren conocer su contexto de llamada lo que determina que el análisis no sea composicional.

En la tesis de licenciatura de Rouaux[RG09] se presenta un nuevo enfoque para la técnica menciona da en [BFGY08], el cual permite abordar el problema de predicción de consumo de forma composicional. Se desarrolla un algoritmo composicional que permite sintetizar un resumen por método que describe los efectos en el *heap*, y se utiliza un análisis para aproximar el tiempo de vida de los objetos que relega precisión en las cotas obtenidas en busca de performance.

En [AGGZ09] se propone un análisis paramétrico de costo para Java secuencial. La idea es traducir un programa en una representación recursiva abstracta, computar apropiadamente invariantes lineales similares a los introducidos en [BFGY08, RG09], y luego calcular sistemas de ecuaciones recurrentes en término de parámetros de entrada. También permite la desasignación de objetos aproximando el tiempo de vida de los objetos usando un análisis de escape.

En [AGGZ10] se presenta una técnica de inferencia de requerimientos de memoria que es paramétrica en la noción de tiempo de vida de un objeto. La idea principal es usar la información de tiempo de vida de los objetos mientras se generan los sistemas de ecuaciones de recurrencia que capturan el uso de memoria en los diferentes estados del programa. Se usa una evaluación parcial para reemplazar un llamado a una relación por su definición combinando las reglas correspondientes en la misma ecuación. Ambas técnicas tiene características en las llamadas a los métodos que hacen que no resulten ser un análisis composicional. Además no aborda características como la de polimorfismo.

En [HH10] se extiende el trabajo realizado en [HJ03] a cotas polinomiales. Se utiliza un análisis automático para deducir cotas de los programas funcionales sin realizar anotaciones extras, siempre y cuando se provea un grado máximo de los polinomios que acotan. El análisis es genérico y puede obtener buenas cotas para el espacio del *heap*, de la pila y el tiempo de uso. Sin embargo el análisis es propuesto para programas funcionales y no para lenguajes orientados a objetos.

En [JHLH10] se describe un nuevo análisis estático para determinar funciones que acoten superiormente al uso cuantitativo de recursos para programas recursivos, polimórficos y posiblemente con aliassing. Utiliza una variante de la técnica de amortizaje manual de análisis de costos de Tarjan [Tar85]. El aspecto novedoso del trabajo es que trata directamente con funciones de polimorfismo sin necesitar de transformaciones de código que puedan alterar el uso de recursos. Sin embargo las cotas se

encuentran restringidas a sumas de polinomios, ya que por ejemplo $n.m$ se representa como $n^2 + m^2$.

Por otro lado, en [GMC09] se presenta una técnica para computar cotas simbólicas en el número de instrucciones ejecutadas por un método en términos de entradas escalares y funciones cuantitativas definidas por el usuario de las estructuras de entrada. La idea es instrumentar el programa con contadores que facilite a la herramienta de generación de invariantes lineales el cálculo individual de cotas lineales. Estas cotas son luego compuestas en conjunto para generar cotas no lineales.

En [GZ10] se presenta una técnica diferente para el cálculo de cotas en ciclos. La idea es generar un sistema de transiciones que describan las relaciones entre valores de las variables del programa en un punto de control y sus valores en el siguiente punto de visita. Luego se computa una función variante para cada transición del sistema usando una técnica de pattern-matching que se basa en hacer algunas consultas que pueden ser resueltas usando un SMT solver. Finalmente todas las cotas son combinadas usando reglas de composición. Si bien el trabajo plantea una solución a programas secuenciales y no recursivos, no trata con el contexto de procedimientos recursivos. Además, tanto esta técnica como la presentada en [GMC09] no consideran la liberación de memoria ya que sirven para analizar el número de visitas a una instrucción, por lo que resultan más imprecisas que la planteada en este trabajo.

9.3. Trabajos a futuro

Soportar métodos recursivos continúa siendo una asignatura pendiente. Si bien tanto el algoritmo como la instanciación presentada permiten incluir cierto tipo de métodos recursivos, estos dependen de la especificación del método por parte del programador. Para soportar patrones más generales de recursión se podría utilizar el enfoque presentado en [AGGZ09, AGGZ10] basado en ecuaciones recurrentes.

Otro aspecto posible sobre el cual se puede trabajar consiste en realizar la integración con alguna herramienta que permita extraer invariantes especificados en el código usando algún lenguaje de especificación como, por ejemplo, JML [LLP+00]. En conjunto con la integración de herramientas que generen invariantes de forma automática, como la presentada en [Gar05], aumentaría la precisión y usabilidad de la herramienta. Además facilitaría el análisis de consumo de programas complejos. En particular, la extracción de invariantes escritos en JML es una característica deseable en si misma pues permite validar la correctitud de los invariantes provistos. La especificación a su vez puede ayudar a determinar la relación entre los parámetros formales de un método y los reales. Para algunas estructuras de datos puede tomarse algún enfoque automático, como el utilizado en [Gar05], pero para estructuras complejas esto no es trivial. Una posibilidad es asistir a las herramientas automáticas especificando la relación dentro del código del programa.

La tarea de generar un lenguaje de especificación de consumo general constituye un trabajo a futuro. Esto permitiría escribir resúmenes en un formato estándar y facilitaría la interacción entre distintas instanciaciones del algoritmo, así como también con otras herramientas capaces de trabajar con estos resúmenes. Por ejemplo, podría ser la entrada para un verificador capaz de determinar si la información inferida es correcta.

Otra aspecto para dedicar esfuerzo consiste en mejorar la precisión o performance de las alternativas propuestas para abordar el problema de las invocaciones polimórfica. Es posible trabajar sobre la performance de la alternativa de coloreo de nodos generando una estrategia de consulta a la calculadora o almacenando y reutilizando los residuales totales calculados. Por otro lado es posible mejorar la precisión de la alternativa de agrupamiento de nodos trabajando sobre nuevas estrategias. A su vez es posible buscar nuevos métodos para abordar esta problemática.

Por último, queda abierta la posibilidad de buscar nuevas instanciaciones para el algoritmo descripto en la Sección 4. Se podría buscar un análisis de escape más preciso que el utilizado en [RG09] que no sufra los problemas presentados en este trabajo ante la presencia de polimorfismo, así como también investigar la utilización de distintas librerías para implementar el CT y SA a fin de poder superar las limitaciones de trabajar con invariantes lineales.

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.

[AGGZ10] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In *ISMM '10: Proceedings of the 2010 international symposium on Memory management*, pages 121–130, New York, NY, USA, 2010. ACM.

[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.

[BFGY08] Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 141–150, New York, 2008. ACM.

[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.

[Bla03] Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, November 2003.

[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.

[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.

[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.

[ECGN99] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE99*, pages 213–224, 1999.

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

[Gar07] Diego Garbervetsky. *Parametric specification of dymamic memory utilization.* PhD thesis, DC, FCEyN, UBA, November 2007.

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

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 127–139, New York, NY, USA, 2009. ACM.

[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.

[GZ10] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 292–304, New York, NY, USA, 2010. ACM.

[HH10] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential programs. *Programming Languages and Systems*, 1480, 2010.

[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.

[JHLH10] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. *ACM*, 2010.

[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.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[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.

[RG09] Martín Rouaux and Diego Garbervetsky. Predicción paramétrica de requerimientos de memoria: Especificación modular. Master's thesis, Departamento de Computación. FCEyN. UBA., August 2009.

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

[SR05a] Alexandru Salcianu and Martin Rinard. A combined pointer and purity analysis for Java programs, January 2005.

[SR05b] Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for Java programs, 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.

[Tar85] R.E. Tarjan. Amortized computational complexity. In *SIAM Journal on Algebraic and Discrete Methods*, pages 306–318, 1985.

[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.

[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.

[Ver10] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software - ICMS 2010,*. Springer, 2010. Accepted.

[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.

Apéndice A

Uso de las herramientas

Este apéndice presenta los comandos necesarios para utilizar las herramientas de análisis de escape y consumo desarrolladas en esta tesis. Para ejemplificar su uso vamos a considerar los métodos `map` y `test` utilizados como parte de los ejemplos motivacionales.

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.

A.1. Invariantes

El formato de archivo aceptado por el *Proveedor de invariantes* debe ser un archivo (*XML*) con el siguiente formato:

Cada archivo de invariantes es descripto mediante un elemento *XML* de tipo *spec* como se puede ver en el ejemplo A.1.

A su vez, debe contener un elemento de tipo *class* con la propiedad *decl*, la declaración de la clase. Notar que debe contener el paquete entero de la misma.

Cada clase debe contener tantos elementos *method* como métodos se desee especificar. Cada método debe contener la propiedad *decl*, la declaración del método. Notar que debe contener el paquete entero de los parámetros formales que recibe y del tipo que devuelve. A su vez, cada método está compuesto por:

relevant-parameters: Parámetros relevantes del método. Notar que pueden diferir de los parámetros formales del mismo como se explico a lo largo del trabajo.

requires: Precondiciones de los parámetros relevantes del método. No es un elemento obligatorio de la especificación.

call-site: Restricción sobre el call número *offset*. La numeración comienza en 0 y es relativa al número de call dentro del método en el *.jimple* asociado a la clase.

La restricción debe estar dentro del elemento *constraints* y constituye el invariante asociado a la técnica de conteo. Puede contener parámetros relevantes, variables (e.g. para marcar un espacio de iteración) y parámetros formales del método llamado. Notar que en este último caso se debe utilizar el prefijo $\$t.$ y el parámetro formal debe también existir en el método llamado. Las expresiones de consumo especificadas pueden ser o constantes o expresiones en función de los parámetros relevantes del método.

creation-site: Restricción sobre el punto de creación número *offset*. La numeración comienza en 0 y es relativa al número de call dentro del método en el *.jimple* asociado a la clase. La restricción debe estar dentro del elemento *constraints* y constituye el invariante asociado a la técnica de conteo. Puede contener parámetros relevantes y variables (e.g. para marcar un espacio de iteración).

```
<?xml version="1.0" encoding="UTF-8"?>
<spec>
 <class decl="ar.uba.dc.EjMotiv">
  <method decl="ar.uba.dc.List test(ar.uba.dc.List, ar.uba.dc.Trans)">
   <relevant-parameters>list.size, list.elements.maxSize</relevant-parameters
       >
   <requires><![CDATA[list.size > 0]]></requires>
   <requires><![CDATA[list.elements.maxSize > 0]]></requires>
    <call-site offset="3">
       <constraints><![CDATA[1 <= k <= list.size and $t.list.size == list.
            elements.maxSize]]></constraints>
    </call-site>
    <call-site offset="2,4-5">
       <constraints><![CDATA[1 <= k <= list.size]]></constraints>
    </call-site>
  </method>
 </class>
</spec>
```

Listing A.1: Ejemplo invariantes método `test`

A.2. Análisis de escape

La sintaxis del comando para ejecutar el análisis es:

./escape.sh nombreClase

Existe un archivo con parámetros de configuración de nombre ***config.properties***. La herramienta posee diferentes formatos de salida:

- Imagen: Archivos para cada método alcanzable en el programa bajo análisis de tipo *.gif* que contienen el grafo con la información del escape.

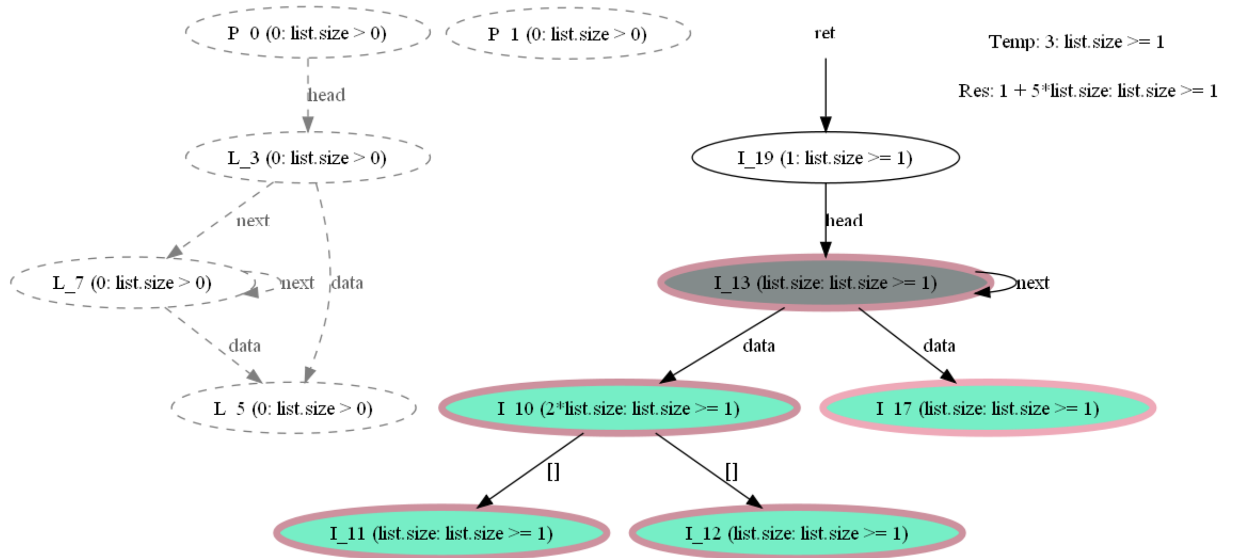


Figura A.1: Salida análisis consumo para el método `map`

- XML: Archivos para cada método alcanzable en el programa bajo análisis de tipo *.xml* que contienen el grafo con la información de escape serializado a formato *XML*. Dado que los mismos son una serializacíón de los grafos, su formato es poco relevante y no se detalla.

A.3. Análisis de consumo

La herramienta toma como entrada para cada método el archivo de tipo *XML* generado por el análisis de escape y utiliza como entrada opcional el archivo *<clase>.spec* descripto anteriormente. La sintaxis del comando es:

./memory.sh nombreClase

La herramienta busca en el directorio *target/clase/escape* los archivos con la información de escape de cada método y en *target/clase/invariants* los archivos *clase.spec* para cada clase analizada. En caso que el análisis de escape no se haya realizado se corre en esta instancia. El archivo *test.spec* 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.

Existe un archivo con parámetros de configuración de nombre ***config.properties***. La herramienta posee diferentes formatos de salida:

- Reporte: Archivo de nombre *index.html* que contiene la descripción de consumo de los métodos alcanzables por el programa bajo análisis.

- Imagen: Archivos para cada método alcanzable en el programa bajo análisis de tipo *.gif* que contienen el grafo decorado con la información del consumo (ver Figura A.1).

- XML: Archivos para cada método alcanzable en el programa bajo análisis de tipo *.xml* que contienen el grafo decorado con la información del consumo serializado a formato *XML*. Dado que los mismos son una serializacíón de los grafos, su formato es poco relevante y no se detalla.

Índice de figuras

1.1. Componentes centrales de la solución. 14

4.1. Algoritmo para construir resúmenes de métodos 35
4.2. Binding inter-procedural de resúmenes de métodos 36
4.3. Binding inter-procedural de resúmenes de métodos considerando invarianza de ciclo 44

5.1. Resultados análisis Salcianu-Rinard para llamadas de map 50
5.2. Resultado análisis map 51
5.3. Análisis inter-procedural primer llamada a ListItr.next 52
5.4. Análisis inter-procedural segunda llamada a ListItr.next 53
5.5. Reglas inter-procedural Salcianu-Rinard 55
5.6. Análisis testMultiply variando valor de k 58

6.1. Resultado análisis escape map polimórfico 72
6.2. Algoritmo Particiones a un único residual 75
6.3. Resultado análisis escape map polimórfico con coloreo 76
6.4. Resultado análisis escape map polimórfico con agrupando 77

8.1. Crecimiento %error en [RG09] al aumentar el valor de nI 91

A.1. Salida análisis consumo método map 106

Índice de cuadros

5.1. Función de transferencia de Rinard para las instrucciones relevantes . 54
5.2. Procedimiento *process_load*. 54

8.1. Resumen experimentos realizados. 87
8.2. Resumen de consumo por método para los ejemplos motivacionales. . 88
8.3. Resumen de consumo por método para MST. 89
8.4. *MemReq* y error para el consumo de MST. 90
8.5. Resumen de consumo por método para EM3D. 90
8.6. *MemReq* y error para el consumo de EM3D. 91
8.7. Resumen de consumo por método para JLayer. 92
8.8. *MemReq* y error para el consumo de JLayer. 93