Tesis de Licenciatura en Ciencias de la Computación

Obtención de cotas del consumo de memoria requerido para ejecutar un método bajo el modelo de memoria por alcance a través de bases de Bernstein.

Federico Javier Fernández - federico.fernandez@gmail.com - L.U. 554/01

Director: Diego Garbervetsky - diegog@dc.uba.ar

Codirectora: Flavia Bonomo - flavia@bonomo.com.ar

1 de septiembre de 2006

Resumen

La comunidad de desarrollo de sistemas embebidos y de tiempo real está inclinándose cada vez mas hacia el uso de lenguajes de programación orientados a objetos, como Java. Este hecho genera nuevos desafíos en la investigación y desarrollo.

Un aspecto particularmente problemático en entornos de tiempo real y embebidos es el manejo automático de memoria. La evaluación de los requerimientos cuantitativos de memoria es inherentemente difícil. A raíz de esta interesante problemática, se plantean trabajos para conseguir una cota superior paramétrica sobre la cantidad de memoria dinámicamente reservada.

Dadas estas técnicas surge, como problema derivado, la búsqueda de la cantidad de memoria requerida para ejecutar un determinado método sabiendo que existe un administrador de memoria que se ocupa de colectar los objetos ya no referenciados.

Es decir, dado un método "m" queremos obtener una cota superior de la cantidad de memoria que se requerirá para su ejecución. Esto se traduce en un interesante problema de optimización no lineal que debe ser resuelto simbólicamente y bajo restricciones de performance muy ajustadas.

Nuestra solución se basa en la aplicación de la técnica de transformación a la base de Bernstein de la cual hicimos la primera implementación existente, aplicándola a nuestro problema con éxito.

Agradecimientos

Haber terminado esta tesis, además de marcar el fin de este trabajo, significa para mi la finalización de la carrera. Por lo tanto, creo que este es un buen momento para agradecer a las personas que estuvieron relacionadas a mi vida académica y personal.

En el plano académico quiero destacar:

- a Diego que desde el primer momento hizo todo lo posible para que trabajemos juntos y que entienda el problema que quería resolver para me dé cuenta lo interesante que era. Desde que lo conocí en sus buenísimas clases de Algoritmos I me pareció una persona para tener cerca, y a medida que pasó el tiempo conocí su divertida personalidad. Estuvo muy bueno trabajar durante este año juntos.
- la colaboración de Philippe, quién nos orientó en el camino hacia la resolución, nos dio la posibilidad de realizar un trabajo original y útil y nos acompañó durante todo el desarrollo con una humildad y motivación increíbles. Todavía nos sorprendemos ante sus reportes y sus comentarios sobre el trabajo del grupo siempre valorando mucho el trabajo de todos, aún siendo un científico de su nivel.
- a Flavia, que todo el tiempo nos iluminó con sus explicaciones matemáticas, sus correcciones detalladas y su organización del trabajo.
- a Cynthia, que me acompañó durante toda la carrera, cursando y estudiando juntos durante estos años; ayudándonos mutuamente ante cada trabajo práctico o parcial.
- a Víctor por su interés en mejorar el trabajo.
- a todos los chicos que pasaron por los grupos de trabajos prácticos (en especial Ale y Lean), con los que compartimos largas horas intentando hacer andar las cosas (usando programación Vodoo si era necesario).
- a los buenos docentes que tuve durante la carrera, que se interesaron por transmitir algo a los alumnos con humildad.

En el plano personal no puedo dejar de agradecer a:

- mis viejos y mi hermano por haberme dado todo para que pueda hacer mi vida sin preocupaciones.
- los chicos de todos@ con los que cursamos y compartimos muchísimos momentos divertidos (y tardes de interminables cadenas de mails).
- mis amigos que me aguantaron durante todos estos años (especialmente Mariano)

Un párrafo especial merece mi novia bonita. Conocerla fue cambiar la forma en que veo el mundo y lograr estar cerca de ella fue un sueño cumplido. Es increíble que una personita pueda hacer que cosas tan simples sean tan felices. No puedo dejar de agradecer infinitamente que me haya acompañado en estos años.

Gracias.... totales.

Índice general

Capítulo 1

Introducción

La comunidad de desarrollo de sistemas embebidos y de tiempo real está inclinándose cada vez más hacia el uso de lenguajes de programación orientados a objetos, como Java. Este hecho genera nuevos desafíos en la investigación y desarrollo.

Un aspecto particularmente problemático en entornos de tiempos real y embebidos es el manejo automático de memoria. La evaluación de los requerimientos cuantitativos de memoria es inherentemente difícil. De hecho, encontrar una cota superior finita del consumo de memoria es indecidible [Ghe]. Este es un gran problema debido a que los sistemas embebidos tienen (en la mayoría de los casos) restricciones de memoria ajustadas o son aplicaciones críticas que no pueden quedarse sin memoria en la ejecución.

A raíz de esta interesante problemática, el trabajo [BGY06] plantea una técnica original para computar una cota superior paramétrica sobre la cantidad de memoria dinámicamente reservada por programas orientados a objetos en lenguajes similares a Java.

En dicho trabajo se presenta una técnica para analizar las reservas de memoria dinámica realizadas por un método. La técnica consiste en, dado un método m con parámetros $p_1, \cdots, p_k$, obtener una expresión sobre los parámetros que sobre aproxima la cantidad de memoria reservada durante su ejecución.

A grandes rasgos, esto se logra de la siguiente manera:

- En primer lugar, se busca un invariante que describe la relación entre las variables del programa en cada sentencia que reserve memoria.
- Luego, la cantidad de memoria consumida está basada en el número de puntos enteros que satisfacen el invariante. Este número es dado en forma paramétrica como un polinomio donde las incógnitas son los parámetros de entrada del método.
- Finalmente, los polinomios deben ser evaluados en los valores de los parámetros de entrada del programa (o método) para obtener la cota.

Dada esta técnica surge, como problema derivado, la búsqueda de la cantidad de memoria requerida para ejecutar un determinado método.

Es decir, dado un método m queremos obtener una cota superior de la cantidad de memoria que se requerirá para su ejecución. Será de gran ayuda contar con el método recién descripto para obtener el uso de memoria por método pero será necesario enfrentar los inconvenientes que surgen de comenzar a considerar las llamadas (directas o indirectas) a otros procedimientos siguiendo el flujo de ejecución del programa.

Dichas llamadas tendrán sus propios parámetros que se encadenarán con los parámetros iniciales, y darán lugar a distintos usos de memoria en cada una de las llamadas derivadas.

Así, se plantea un interesante problema de optimización que, adicionalmente, debe ser resuelto muy eficientemente (ya que en los entornos de sistemas embebidos y de tiempo real es estrictamente necesario). Esta resolución permitiría aplicaciones tales como la carga dinámica de aplicaciones con verificación previa de sus cotas de memoria.

Resolver este problema es el objetivo de esta tesis, utilizando la técnica de conteo de memoria y la técnica de las bases de Bernstein para realizar la optimización que implementaremos por primera vez (de manera original).

1.1. Estado del arte

El problema de la estimación dinámica de memoria ha sido estudiado para lenguajes funcionales en [HJ03, HP99, USL03]. El trabajo en [HJ03] infiere estáticamente, por derivación de tipos y programación lineal, expresiones lineales que dependen de los parámetros de las funciones. La técnica es planteada para programas funcionales corriendo bajo un modelo de memoria especial. Las expresiones obtenidas son restricciones lineales en los tamaños de varias partes de los datos. En [HP99] una variante de ML (un lenguaje funcional -estricto- matemáticamente puro, es decir, con instrucciones con sintaxis puramente matemática) es propuesta junto con un sistema de tipos basado en la noción de "sized types" [HPS96], de manera que los programas bien tipados están probados en cuanto a ejecutarse dentro de cotas de memoria dadas. La técnica propuesta en [USL03] consiste en, dada una función, construir una nueva función que simbólicamente imita las reservas de memoria de la original. Esta nueva función debe ser ejecutada sobre una valuación de parámetros para obtener una cota de memoria para dicha valuación. La evaluación de la función de cota podría no terminar, aún si la original terminaba.

En cuanto a lenguajes imperativos orientados a objetos, se han propuesto soluciones en [Ghe, CKQ+05, CHQR05]. La técnica de [Ghe] manipula expresiones aritméticas simbólicas con incógnitas que no son necesariamente variables del programa, agregadas por el proceso de análisis para representar, por ejemplo, iteraciones de los ciclos. La fórmula

resultante debe ser evaluada sobre las incógnitas restantes para obtener una cota superior. No hay disponibles resultados de pruebas de benchmark para saber el impacto de esta técnica en la práctica. Por otra parte, el método propuesto en [CKQ+05, CHQR05] se basa en un sistema de tipos y anotaciones, similar a [HP99]. No obtiene cotas de memoria, pero estáticamente chequea que las anotaciones de tamaño (en forma de fórmulas de Presburger) son verificadas. Es responsabilidad del programador revisar el tamaño de las restricciones, las cuales son de hecho lineales.

Por otra parte, el problema específico de la estimación de la memoria necesaria para ejecutar un método no posee soluciones simbólicas conocidas hasta el momento (para lenguajes imperativos y con ciertas características que más adelante mencionaremos).

Analizando el problema, conocimos una técnica matemática desarrollada en [CT04] que podría servir para resolver este problema. El proceso se basa en la transformación del polinomio original (que representa el consumo de memoria) en una base particular, fuertemente relacionada con el conjunto de invariantes que afecta ese camino. De esta manera, se obtienen cotas relacionadas con los coeficientes obtenidos en esta nueva base. Las cotas pueden ser exactas o aproximadas (conservativamente).

Originalmente la técnica fue planteada para realizar optimizaciones dentro de ciclos, con el objetivo de maximizar paralelismo. Lo que se necesita saber en ese caso es si las dependencias de datos entre iteraciones se superponen, lo cual impediría que se puedan realizar paralelizaciones dentro de los ciclos. En el caso de que las cotas del ciclo en cuestión sean lineales no son necesarias técnicas de este tipo, pero cuando se hacen más complicadas estas funciones es necesario utilizar la técnica para obtener cotas sobre las variables inductivas y así comprobar si es posible la paralelización.

Lamentablemente este enfoque no ha sido implementado aún.

1.2. Contribuciones

Teniendo como punto de partida el trabajo realizado en [BGY05a] para la síntesis de memoria de métodos determinados en esta tesis nos proponemos contribuir un modelo para obtener la memoria requerida para ejecutar un método.

Es decir, dado un programa podemos saber cual es el cálculo a realizar para obtener información sobre la cantidad de memoria que se utiliza para determinados parámetros, pero no ha sido desarrollado aún un modelo que considere las cadenas de llamados, las regiones asociadas, y las relaciones entre cada uno de los métodos para obtener una cota general de la ejecución del grafo de llamados a partir de un método.

Además, el método anterior no considera la existencia de un "recolector de basura" (garbage collector) o administrador de regiones. Bajo este nuevo modelo debemos, en principio, adaptar la técnica para calcular cantidades de memoria utilizada que es capturada por este manejador de memoria (y cual escapa a su alcance). Teniendo esto en cuenta, se debe realizar una optimización de la sumatoria de las distintas cantidades de

memoria utilizadas (y que son capturadas) a través del flujo del programa dado por el grafo de llamados.

Como veremos luego, el cálculo de las fórmulas resultantes de este modelo implican la resolución de un interesante problema de maximización, que dado el contexto de aplicación tiene que cumplir fuertes restricciones en cuanto a que su cálculo debe ser simbólico y eficiente.

Con el modelo planteado, aplicaremos una técnica basada en bases de Bernstein. Esta aplicación será otro importante aporte que requiere también de implementación para su automatización.

Queremos destacar que las soluciones descriptas se tratan de aproximaciones (siempre sobre-aproximaciones) de las cotas a hallar, y como parte de estas tesis exploraremos el balance necesario para que las ejecuciones se den en tiempos razonables y que la aproximación sea lo suficientemente útil (desde el punto de vista de lo ajustada a la realidad).

Finalmente, la principal contribución de esta tesis es la implementación de la técnica de Bases de Bernstein desarrollada por Philippe Clauss en [CT04] con grandes aplicaciones en el análisis de programas y optimización. Esta implementación trajo consigo avances teóricos que fueron necesarios para cerrar la técnica. Algunos de estos avances serán vistos en la sección 5.2.6 cuando hablemos de los métodos que desarrollamos para obtener máximos coeficientes de Bernstein de manera simbólica (por ejemplo, a través de aplicaciones recursivas).

1.3. Estructura de la tesis

El documento se estructura de la siguiente manera:

- En el capítulo 2 se presentan algunos resultados preliminares que serán útiles para los conceptos presentados en el resto del documento.
- En el capítulo 3 presentamos la técnica para sintetizar los requerimientos de memoria, planteando sobre el final el problema que nos motiva.
- En el capítulo 4 describimos las bases de Bernstein y esbozamos una aproximación a la resolución de nuestro problema a través de ésta técnica.
- En el capítulo 5 nos encargamos de resolver el problema de optimización utilizando todo lo visto en los capítulos previos y comentamos otras aplicaciones de la técnica.
- En el capítulo 6 se describe en profundidad los detalles de la implementación realizada.
- En el capítulo 7 presentamos las pruebas realizadas y una discusión sobre los resultados obtenidos.

- Luego, se evalúa el trabajo a futuro y las conclusiones.
- Por último, presentamos un apéndice con algunos cálculos detallados de complejidad.

Capítulo 2

Conceptos preliminares

2.1. Métodos de programación para optimización

Un problema de programación lineal es un problema de maximización que tiene la siguiente forma:

$$z = \text{máx}\{cx : x \in P\} \tag{2.1.1}$$

Donde $P = \{x : Ax \leq b\}$ es un poliedro y $c \in \Re^n$.

Un problema de programación entera es un problema de maximización que tiene la siguiente forma:

$$z = \text{máx}\{cx : x \in P,\ x \in Z^n\} \tag{2.1.2}$$

Donde $P = \{x : Ax \leq b\}$ es un poliedro y $c \in \Re^n$.

El problema de programación entera es NP-hard [GW88].

Un problema de programación no lineal es un problema de maximización donde la función objetivo no necesariamente tiene por que ser lineal (por ejemplo, un polinomio).

2.2. Complejidad algorítmica

Un problema algorítmico $\pi(I, Q)$ consta de un conjunto I de todas las posibles entradas para el problema, llamado el conjunto de instancias, y de una pregunta Q sobre esas instancias. Resolver uno de estos problemas consiste en desarrollar un algoritmo cuya entrada es una instancia del problema y cuya salida es una respuesta a la pregunta del problema.

Un problema es de optimización cuando lo que se busca a través de la pregunta es la solución óptima para el problema formulado. Por ejemplo, "dado un un plan de producción, ¿cuál es el máximo rendimiento posible?"

Diremos que un algoritmo es polinomial cuando el número de operaciones que efectúa está acotado por una función polinomial en el tamaño de su entrada. Si el tamaño de la entrada es n y la función polinomial es $f(n)$, decimos que el algoritmo tiene complejidad $O(f(n))$. Los problemas de decisión para los que existen algoritmos polinomiales constituyen la clase P y son llamados polinomiales.

Un problema de decisión es no-determinístico polinomial cuando cualquier instancia que produce respuesta SI posee una comprobación de correctitud (también llamada certificado) verificable en tiempo polinomial en el tamaño de la instancia. Estos problemas de decisión pertenecen a la clase NP.

Claramente, P $\subseteq$ NP. Sin embargo, no se sabe si esta inclusión es estricta: uno de los principales problemas abiertos en informática teórica es saber si P $\neq$ NP.

Un problema de decisión pertenece a la clase co-NP cuando cualquier instancia que produce respuesta NO posee un certificado polinomial en el tamaño de la instancia.

Una definición esencial en la teoría de complejidad es la definición de problema NP-completo. Un problema de decisión π pertenece a la clase NP-completo cuando se satisfacen las siguientes condiciones:

- $\pi \in$ NP.
- Para todo problema $\pi' \in$ NP, existe una transformación polinomial de π' en π.

Un problema de decisión π pertenece a la clase NP-hard cuando se satisface la siguiente condición:

- Para todo problema $\pi' \in$ NP, existe una transformación polinomial de π' en π.

La teoría de NP-completitud fue iniciada por Cook en 1971 [Coo71]. Allí probó que el problema de satisfactibilidad de la lógica matemática es NP-completo, en un resultado que se conoce como el Teorema de Cook. Primero Karp [Kar72], y tiempo después Garey y Johnson [GJ79], presentaron largas listas de problemas NP-completos en el campo de la combinatoria, la lógica, la teoría de conjuntos, la teoría de grafos y otras áreas de la matemática discreta.

La técnica standard para probar que un problema π es NP-completo es la siguiente: elegir en forma apropiada un problema π' que ya sabemos que es NP-completo y luego probar que $\pi \in$ NP y que π' es transformable polinomialmente en π. Si sólo probáramos esta segunda parte, habríamos probado que el problema π es NP-hard.

No se conoce ningún algoritmo polinomial para resolver un problema NP-completo. Surge de la definición de NP-completitud que si se encontrara un algoritmo polinomial para un problema en esta clase, todo problema en NP sería polinomial (y estaría probado, en consecuencia, que P = NP).

Similarmente, se define una clase de complejidad para los problemas cuya respuesta es un número natural, en la cual entran la mayoría de los problemas de enumeración relacionados con problemas NP-completos y algunos problemas de aritmética como por ejemplo calcular la permanente de una matriz [Val79]. Un problema π está en la clase #P si para toda instancia D de π, cuya respuesta es n, la respuesta posee una comprobación de correctitud (también llamada certificado) verificable en tiempo $n.p(|D|)$, donde p es un polinomio que depende de π.

2.3. Teoría poliedral

Un poliedro es un conjunto $P = \{x \in \Re^n : Ax \leq b\}$, donde $A \in \Re^{m \times n}$ y $b \in \Re^m$.

Dado un conjunto finito de puntos $K = \{v_1, \ldots, v_t\} \subseteq \Re^n$, la cápsula convexa de K, llamada $conv(K)$, es la intersección de todos los conjuntos convexos que incluyen a K.

Un conjunto de puntos en $\Re^n$ es convexo si dados dos puntos x e y en el conjunto, el segmento determinado por ellos pertenece al conjunto.

Una combinación convexa de los puntos de K es un punto v de la forma $v = \sum_{i=1}^{t} \alpha_i v_i$ con $\alpha_i \geq 0$ y $\sum_i \alpha_i = 1$. Una combinación convexa se dice *propia* cuando al menos dos de los α_i son no nulos.

Es un resultado conocido que

$$conv(K) = \Big\{ \sum_{i=1}^{t} \alpha_i v_i : \alpha_i \geq 0 \text{ y } \sum_i \alpha_i = 1 \Big\}$$

es decir, la cápsula convexa de un conjunto K es el conjunto de todas las combinaciones convexas de los puntos de K.

Dado un conjunto finito de puntos $K = \{v_1, \ldots, v_t\} \subseteq \Re^n$, el cono determinado por K es $cono(K) = \Big\{ \sum_{i=1}^{t} \alpha_i v_i : \alpha_i \geq 0 \Big\}$.

El poliedro P es acotado si y sólo si K' es vacío.

Definición. Los puntos $v_1, \ldots, v_k \in \Re^n$ son afinmente independientes si la única solución de

$$\sum_{i=1}^{k} \alpha_i v_i = 0 \qquad \sum_{i=1}^{k} \alpha_i = 0$$

es $\alpha_i = 0$ para $i = 1, \ldots, k$.

El punto v es un extremo del poliedro P si v no puede ser escrito como combinación convexa de dos o más puntos en P.

Un poliedro se dice entero cuando sus extremos son enteros.

2.4. Cantidad de soluciones de un conjunto de restricciones

Habiendo repasado las definiciones geométricas necesarias, pasaremos a describir el método utilizado para contar puntos enteros dentro de conjuntos de restricciones, lo cual será útil al momento de realizar las estimaciones de memoria.

Sea $\mathcal{I}$ un conjunto de restricciones sobre un conjunto de variables enteras $V = W \uplus P$ donde P representa un conjunto de variables distinguidas (llamadas parámetros) y W el resto de las variables que aparecen en las restricciones.

$\mathcal{C}(\mathcal{I}, P)$ denota una expresión simbólica sobre P que provee el *número de soluciones enteras* en $\mathcal{I}$ para el conjunto de variables W, asumiendo que P tiene valores fijos. Más precisamente:

$$\mathcal{C}(\mathcal{I}, P) = \lambda\vec{p}.(\ \#\{\ \vec{w} \mid \mathcal{I}\ [W/\vec{w}, P/\vec{p}\ ]\ \}\ )$$

Existen varias técnicas para obtener estas expresiones simbólicas [Cla96, Fah98].

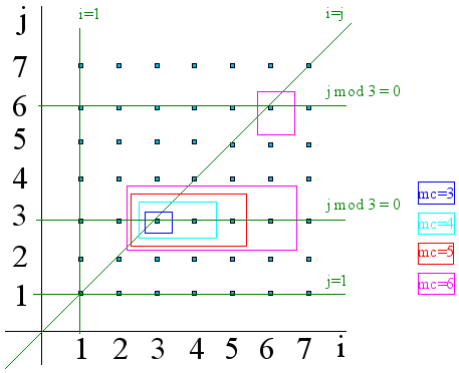
En este trabajo, usaremos la técnica descripta en [Cla96], en la cual las expresiones resultantes son polinomios (polinomios de Ehrhart [Ehr77]) los cuales poseen coeficientes que son funciones periódicas de los parámetros.

Veamos, como un ejemplo de lo expuesto, la siguiente tabla que muestra algunos conjuntos de restricciones lineales y sus polinomios de Ehrhart correspondientes:

$\mathcal{I}$	W	P	$\mathcal{C}(\mathcal{I}, P)$
$\{k = mc, 1 \leq i \leq k\}$	$\{k, i\}$	$\{mc\}$	mc
$\{k = mc, 1 \leq i \leq k, 1 \leq j \leq i, n = i, j \bmod 3 = 0\}$	$\{k, i, j, n\}$	$\{mc\}$	$\frac{1}{6}mc^2 - \frac{1}{6}mc + per(mc, [0, 0, -\frac{1}{3}])$
$\{n = 2mc\}$	$\{n\}$	$\{mc\}$	1

donde para una incógnita x y un vector $a = [a_0, \ldots, a_{k-1}]$ de elementos, $per(x, a)$ denota el elemento a_i con $i = x \bmod k$. El siguiente diagrama ilustra el resultado de la evaluación del polinomio de la segunda fila de la tabla anterior:

mc	$\frac{1}{6}mc^2 - \frac{1}{6}mc + per(mc, [0, 0, -\frac{1}{3}])$
1	0
2	0
3	1
4	2
5	3
6	5



Capítulo 3

Sintetizando el uso de memoria

Con el objetivo de definir y acotar el alcance del problema a resolver, en este capítulo presentaremos la técnica descripta en [BGY05b] para obtener las fórmulas que representan la estimación conservativa de memoria, que será central en el cálculo de la memoria requerida para un método.

En primer lugar, mostraremos como adaptar la técnica de conteo previamente tratada para ser utilizada con solicitudes de memoria. Luego, mostraremos como computar la cantidad de memoria reservada por un método.

A lo largo de esta sección utilizaremos un ejemplo que presentamos a continuación, en la Figura 3.1.

```
    void m0(int mc) {
1:    Ref0 h = new Ref0();
2:    Object[] a = m1(mc);
3:    Object[] e = m2(2*mc,h);
    }
    Object[] m1(int k) {
1:    int i;
2:    Ref0 l = new Ref0();
3:    Object[] b = newA Object[k];
4:    for(i=1;i<=k;i++)  {
5:        b[i-1] = m2(i,l);
      }
6:    Object[] c = newA Integer[9];
7:    return b;
    }
class Ref0 {
   public Object ref;
}
```

```
    Object[] m2(int n, Ref0 s) {
1:    int j;
2:    Object c,d,e;
3:    Object[] f = newA Object[n]

4:    for(j=1;j<=n;j++) {
5:      if(j % 3 == 0) {
6:        c = newA Integer[j*2+1];
        }
        else {
7:        c = new Integer(0);
        }
8:      d = newA Integer[4];
9:      f[j-1] = c;
      }
10:   e = newA Integer[1];
11:   s.ref = e;
12:   return f;
     }
```

Figura 3.1: Ejemplo

3.1. Notación para programas

Definimos un programa como un conjunto $\{m_0, m_1, \ldots\}$ de métodos.

Un método posee una lista P_m de parámetros ($\vec{p_m}$ denotará los argumentos del método cuando m es llamado por otro método m') y una secuencia de sentencias. Por simplicidad, comenzaremos asumiendo que los parámetros del método serán de tipo entero. Además, asumiremos que no hay conflicto de nombres incluyendo parámetros formales, nombres de variables locales y globales. Por otro lado, la recursión no está permitida y, si está presente, será eliminada usando transformaciones del código.

Cada sentencia del programa está identificada con una ubicación de control $Label =_{def} Method \times \mathbb{N}$ (un método y una posición dentro del método) que caracterizan de manera única la sentencia a través de un mapping stm ($stm : Label \rightarrow Statement$).

3.2. Representación del estado de un programa

En pocas palabras, el estado de un programa en tiempo de ejecución está dado por los valores de las variables, la posición de control y la pila de llamados.

La estimación de consumo de memoria es básicamente cuestión de contar la máxima cantidad de veces que las sentencias de creación de objetos son ejecutados en una corrida del programa. La idea es encontrar una abstracción que sea capaz de describir (conservativamente) los estados del programa y de tal manera que sea posible aplicar la técnica de conteo antes mostrada.

Para describir los estados de un programa utilizamos una abstracción donde separamos la parte de control y la pila de llamados. Esto es, los *estados de control* son pares (π, l), denotados como $s = \pi.l$, donde $l = (m, n) \in Label$ es la ubicación de la sentencia y $\pi \in Label^*$ es el camino al método m en el árbol de llamados.

Un *invariante* para un estado de control s es una aserción sobre las variables del programa (locales y globales) que se mantiene válida siempre que el estado es alcanzado en cualquier corrida.

Dado un programa comenzando en un método m y un estado de control $s = \pi$ para dicho programa (por ejemplo $\texttt{first}(\pi.l) = (m, n)$), $\mathcal{I}_s^m$ denota un predicado invariante sobre las variables del programa, para el estado de control s.

Luego, el par $< s, \mathcal{I}_{cs}^m >$ es una aproximación conservativa de los posibles estados del programa en la ubicación l con la pila π en cualquier corrida comenzando con una invocación del método m.

La segunda columna de la tabla 3.1 muestra los invariantes lineales para algunos estados de control para un programa comenzando en el método $m0$.

3.3. Cantidad de visitas de un estado de control

Como fue observado, un invariante para un estado de control cs restringe la posibilidad de valores de cualquiera de las corridas que se abstrae mediante s. Esto, junto con el hecho que un estado de control también define la configuración de la pila de llamados, implica que el conteo del número de soluciones enteras de dicho invariante arroja una expresión que sobre aproxima el número de veces que un estado concreto (cuya abstracción es s) es alcanzado en una corrida que comienza en el método analizado.

La tercera columna de la tabla 3.1 muestra los polinomios de Ehrhart que cuentan el número de soluciones para algunos estados de control para un programa que comienza en el método $m0$.

cs	$\mathcal{I}^{\mathbf{m0}}_{\mathbf{cs}}$	$\mathcal{C}(\mathcal{I}^{\mathbf{m0}}_{\mathbf{cs}}, \mathbf{P_{m0}})$
$m0$,2.**m1**,**2**	$\{k = mc\}$	1
$m0$,2.**m1**,**3**	$\{k = mc\}$	1
$m0$,2.**m1**,**6**	$\{k = mc\}$	1
$m0$,2.$m1$,5.**m2**,**3**	$\{k = mc, 1 \leq i \leq k, n = i\}$	mc
$m0$,2.$m1$,5.**m2**,**6**	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n, j \bmod 3 = 0\}$	$\frac{1}{6}mc^2 - \frac{1}{6}mc + per(mc, [0, 0, -\frac{1}{3}])$
$m0$,2.$m1$,5.**m2**,**7**	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n, j \bmod 3 > 0\}$	$\frac{1}{3}mc^2 + \frac{2}{3}mc + per(mc, [0, 0, \frac{1}{3}])$
$m0$,2.$m1$,5.**m2**,**8**	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}$	$\frac{1}{2}mc^2 + \frac{1}{2}mc$
$m0$,2.$m1$,5.**m2**,**10**	$\{k = mc, 1 \leq i \leq k, n = i\}$	mc
$m0$,3.**m2**,**3**	$\{n = 2mc\}$	1
$m0$,3.**m2**,**6**	$\{n = 2mc, 1 \leq j \leq n, j \bmod 3 = 0\}$	$\frac{2}{3}mc + per(mc, [0, -\frac{2}{3}, -\frac{1}{3}])$
$m0$,3.**m2**,**7**	$\{n = 2mc, 1 \leq j \leq n, j \ mod \ 3 > 0\}$	$\frac{4}{3}mc + per(mc, [0, \frac{2}{3}, \frac{1}{3}])$
$m0$,3.**m2**,**8**	$\{n = 2mc, 1 \leq j \leq n\}$	$2mc$
$m0$,3.**m2**,**10**	$\{n = 2mc\}$	1

Cuadro 3.1: Algunos invariantes y polinomios de Ehrhart para $m0$

3.4. Memoria reservada por un punto de creación

Ahora enfocaremos nuestra atención en sentencias que crean nuevos objetos: `new` y `newA`. Asumimos que estas sentencias solamente crean instancias de objetos y los

constructores son llamados de manera separada y tratados como cualquier otra llamada a método. Llamamos *punto de creación*, y lo notamos ms, a un estado de control asociado a las operaciones que solicitan memoria: $ms \in \{\ \pi.l \in Label^{+} \mid stm(l) \in \{\texttt{new T}, \texttt{newA T}[\cdot]\ldots[\cdot]\}\ \}$.

Para calcular la cantidad de memoria reservada por un punto de creación definimos la función $\mathcal{S}$ (ver debajo). Dado un invariante $\mathcal{I}^m_{cs}$ para ese estado de control cs y un método m con parámetros P_m, $\mathcal{S}$ calcula el número de visitas a cs usando el método de conteo y multiplica la expresión resultante por el tamaño del objeto reservado.

Esto es cierto para sentencias *new*. En cambio, en el caso de la creación de arreglos (es decir, $\texttt{newA T}[e_1]\ldots[e_n]$), ésta técnica necesita ser ligeramente adaptada considerando el hecho que un arreglo es una colección de elementos del mismo tipo. De hecho, la sentencia $\texttt{newA T}[e_1]\ldots[e_n]$ crea el mismo número de instancias (y, por lo tanto, reserva la misma cantidad de memoria) que n ciclos anidados de la forma:

$\texttt{for(}\ h_1 = 1;\ h_1 \leq e_1;\ h_1{+}{+}\ \texttt{)}$
$\ldots$
$\quad\texttt{for(}\ h_n = 1;\ h_n \leq e_n;\ h_n{+}{+}\ \texttt{)}$
$\quad\quad\texttt{newA T}[1]$

los cuales poseen un espacio de iteración que puede ser descripto por el invariante $\bigcup_{i=1..n}\{1 \leq h_i \leq e_i\}$.

Por lo tanto, definimos la función $\mathcal{S}$ de la siguiente manera:

$\mathcal{S}(\mathcal{I}^m_{ms},\ P_m,\ \texttt{ms})$ // devuelve una expresion sobre P_m
$\texttt{l = last(ms);}$ // (ms = π.l)
$\texttt{if stm(l)= new T}$
$\quad\texttt{res:=}size(\texttt{T})\cdot\mathcal{C}(\mathcal{I}^m_{ms},\ P_m)\texttt{;}$
$\texttt{else if stm(l)= newA T}[e_1]\ldots[e_n]$
$\quad\texttt{Inv}_{array}\texttt{:=}\ \mathcal{I}^m_{ms} \cup \bigcup_{i=1..n}\{1 \leq h_i \leq e_i\}$
$\quad\texttt{res:=}size(\texttt{T[]})\cdot\mathcal{C}(\texttt{Inv}_{array},\ P_m)\texttt{;}$
$\texttt{end if;}$
$\texttt{return res;}$

donde $size(\texttt{T})$ es una expresión simbólica que denota el tamaño de un objeto del tipo T, y $size(\texttt{T[]})$ es una expresión simbólica que denota el tamaño de una *celda* de un arreglo de tipo T. $\mathcal{C}$ es la expresión que cuenta el número de soluciones enteras de un invariante como fue definido en 2.4.

Como los invariantes lineales son conservativos, $\mathcal{S}$ es, en general, una sobre-aproximación de la memoria reservada.

La tabla 3.2 muestra los polinomios que sobre-aproximan la cantidad de memoria reservada para (algunos) puntos de creación alcanzables desde el método $m0$.

ms	$\mathcal{S}(\mathcal{I}^{\mathbf{m0}}_{\mathbf{ms}}, \mathbf{P_{m0}}, \mathbf{ms})$
$m0$,2.$m1$,2	$size(\texttt{Ref0})$
$m0$,2.$m1$,6	$size(\texttt{Integer[]}) \cdot 9$
$m0$,2.$m1$,5.$m2$,3	$size(\texttt{Object[]}) \cdot (\frac{1}{2}mc^2 + \frac{1}{2}mc)$
$m0$,2.$m1$,5.$m2$,6	$size(\texttt{Integer[]}) \cdot (\frac{1}{9}mc^3 + \frac{1}{2}mc^2 + per(mc, [-\frac{1}{6}, -\frac{1}{6}, -\frac{5}{6}])mc + per(mc, [0, -\frac{4}{9}, -\frac{11}{9}]))$
$m0$,2.$m1$,5.$m2$,7	$size(\texttt{Integer}) \cdot (\frac{1}{3}mc^2 + \frac{2}{3}mc + per(mc, [0, 0, \frac{1}{3}])$
$m0$,2.$m1$,5.$m2$,8	$size(\texttt{Integer[]}) \cdot (2mc^2 + 2mc)$
$m0$,3.$m2$,3	$size(\texttt{Object[]}) \cdot 2mc$
$m0$,3.$m2$,6	$size(\texttt{Integer[]}) \cdot (\frac{4}{3}mc^2 + per(mc, [2, -\frac{2}{3}, \frac{2}{3}])mc + per(mc, [0, -\frac{2}{3}, -\frac{2}{3}]))$
$m0$,3.$m2$,7	$size(\texttt{Integer}) \cdot (\frac{4}{3}mc + per(mc, [0, \frac{2}{3}, \frac{1}{3}])$
$m0$,3.$m2$,8	$size(\texttt{Integer[]}) \cdot 8mc$

Cuadro 3.2: Polinomios de reserva de memoria.

Consideremos por ejemplo el punto de creación $m0$,3.$m2$,8, que corresponde a la sentencia `d = newA Integer[4]` en la línea 8 del método $m2$ cuando es llamado desde $m0$ en la línea 3. El resultado mostrado en la tabla es calculado de la siguiente manera:

$$
\begin{aligned}
&\mathcal{S}(\mathcal{I}^{m0}_{m0,3.m2,8}, mc, m0,3.m2,8) = \\
&\quad = \; size(\texttt{Integer[]}) \cdot \mathcal{C}(\mathcal{I}^{m0}_{m0,3.m2,8} \cup \{1 \leq h \leq 4\}) \\
&\quad = \; size(\texttt{Integer[]}) \cdot \mathcal{C}(\{n = 2mc, 1 \leq j \leq n, 1 \leq h \leq 4\}) \\
&\quad = \; size(\texttt{Integer[]}) \cdot \mathcal{C}(\{1 \leq j \leq 2mc, 1 \leq h \leq 4\}) \\
&\quad = \; size(\texttt{Integer[]}) \cdot 8mc
\end{aligned}
$$

3.5. Memoria reservada por un método

Habiendo mostrado como calcular la cantidad de memoria reservada por un único punto de creación, ahora determinamos cuanta memoria es reservada por una corrida a partir del método m. La técnica básicamente identifica los puntos de creación alcanzables desde ese método, obtiene los invariantes correspondientes, calcula la cantidad de memoria reservada por cada uno y finalmente arroja como resultado la suma de ellos.

MS_m denota el conjunto de puntos de creación alcanzables desde el método m (es decir, caminos $\pi.l$ donde π es un camino en el grafo de llamados a partir de m y $stm(l)$ es una nueva sentencia). Los puntos de creación del ejemplo en la Fig. 3.1 son:

$MS_{m0} = \{$ $m0{,}1$, $m0{,}2.m1{,}2$, $m0{,}2.m1{,}3$, $m0{,}2.m1{,}6$, $m0{,}2.m1{,}5.m2{,}3$, $m0{,}2.m1{,}5.m2{,}6$, $m0{,}2.m1{,}5.m2{,}7$, $m0{,}2.m1{,}5.m2{,}8$, $m0{,}2.m1{,}5.m2{,}10$, $m0{,}3.m2{,}3$, $m0{,}3.m2{,}6$, $m0{,}3.m2{,}7$, $m0{,}3.m2{,}8$, $m0{,}3.m2{,}10$ $\}$
$MS_{m1} = \{$ $m1{,}2$, $m1{,}3$, $m1{,}6$, $m1{,}5.m2{,}3$, $m1{,}5.m2{,}6$, $m1{,}5.m2{,}7$, $m1{,}5.m2{,}8$, $m1{,}5.m2{,}10$ $\}$
$MS_{m2} = \{$ $m2{,}3$, $m2{,}6$, $m2{,}7$, $m2{,}8$, $m2{,}10$ $\}$

La Figura 3.2 muestra el árbol de llamados aumentado con los puntos de creación. Este grafo es automáticamente construido con la herramienta descripta en [BFG+05].

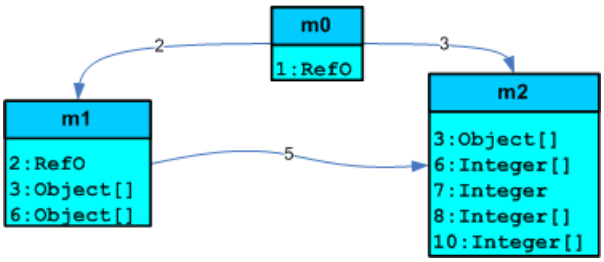


Figura 3.2: Grafo de llamados del Ejemplo y Puntos de Creación

Debemos observar que, dado que no estamos tratando programas recursivos, el número de caminos en el grafo de llamados, y por lo tanto el número de estados de control, es finito.

Ahora, el problema de calcular una cota superior paramétrica de la cantidad de memoria reservada por un método m puede ser reducida a: para todo $ms \in MS_m$, obtener un invariante, calcular la función $\mathcal{S}$ y sumar el resultado que arroja.

La función `computeAlloc` calcula una expresión (en términos de los parámetros del método) que sobre aproxima la cantidad de memoria reservada por un conjunto de puntos de creación.

$$\texttt{computeAlloc(m, MS)} = \sum_{ms \in MS} \mathcal{S}(\mathcal{I}^m_{ms}, P_m, ms)$$

donde $MS \subseteq MS_m$

Dado un método m, el estimador simbólico de la memoria dinámicamente reservada por m se define de lo siguiente manera:

$$\texttt{memAlloc}(m) = \texttt{computeAlloc}(m, MS_m)$$

La tabla 3.3 muestra la expresión calculada para $m0$, $m1$ y $m2$.

memAlloc($m0$)	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{23}{6}mc^2 + (per(mc, [\frac{29}{2}, \frac{71}{6}, \frac{25}{2}]))mc + per(mc, [11, \frac{83}{9}, \frac{79}{9}])\right) + size(\texttt{Integer}) \cdot \left(\frac{1}{3}m^2 + 2mc + per(mc, [0, \frac{2}{3}, \frac{2}{3})\right) + +size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{7}{2}mc\right) + 2 \cdot size(\texttt{Ref0})$
memAlloc($m1$)	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}k^3 + \frac{5}{2}k^2 + (per(k, [\frac{23}{6}, \frac{23}{6}, \frac{19}{6}])k + (per(k, [9, \frac{77}{9}, \frac{70}{9}])\right) + size(\texttt{Integer}) \cdot \left(\frac{1}{3}k^2 + \frac{2}{3}k + (per(k, [0, 0, \frac{1}{3}]) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}k^2 + \frac{3}{2}k\right) + size(\texttt{Ref0})$
memAlloc($m2$)	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{3}n^2 + (per(n, [\frac{16}{3}, \frac{14}{3}, 4])n + (per(n, [2, 1, \frac{2}{3}])\right) + size(\texttt{Integer}) \cdot \left(\frac{2}{3}n + (per(n, [0, \frac{1}{3}, \frac{2}{3}])\right) + size(\texttt{Object[]}) \cdot n$

Cuadro 3.3: Memoria reservada por los métodos $m0$, $m1$, y $m2$

3.6. Aplicaciones a memoria por "scope"

El manejo de memoria por "scope" está basado en la idea de agrupar conjuntos de objetos en regiones que son asociados con el tiempo de vida de una unidad computacional. Por lo tanto, los objetos son recolectados juntos cuando su unidad de ejecución correspondiente finaliza su ejecución.

En virtud de inferir la información de alcance se utilizan técnicas de análisis de punteros y escape (por ejemplo [SR01, Bla99]). En particular, asumimos que, en la invocación de un método una nueva región es creada y que contendrá todos los objetos que son capturados por este método. Cuando termina, la región es limpiada con todos sus objetos.

Un objeto escapa a un método cuando su tiempo de vida es más largo que el de su método. Por lo tanto no puede ser recolectado de manera segura cuando la unidad finaliza su ejecución.

Sea $escape : Method \to I\!P(CreationSite)$ una función que dado un método m devuelve los puntos de creación $\in MS_m$ que escapan a m. Esto es los estados abstractos

$ms = \pi.l$ donde el objeto creado en l escapa a todos los métodos en π [GCYZ04]. Un objeto es capturado por el método m cuando puede ser recolectado de manera segura al final de la ejecución de m. Sea $capture : Method \rightarrow \mathbb{P}(CreationSite)$ una función que dado un método m devuelve los puntos de creación $\in MS_m$ que son capturados por m. Esto es, los estados abstractos $ms = \pi.l$ donde los objetos creados en l escapan todos los métodos en $\pi.l$ excepto m [GCYZ04].

Por ejemplo, para nuestro ejemplo en Fig. 3.1 tenemos:

$$
\begin{aligned}
escape(m0) &= \{\} \\
escape(m1) &= \{m1{,}3, m1{,}5.m2{,}3, m1{,}5.m2{,}6, m1{,}5.m2{,}7\} \\
escape(m2) &= \{m2{,}3, m2{,}6, m2{,}7, m2{,}10\} \\
capture(m0) &= \{m0{,}1, m0{,}2.m1{,}3, m0{,}2.m1{,}5.m2{,}3, \\
&\quad\; m0{,}2.m1{,}5.m2{,}6, m0{,}2.m1{,}5.m2{,}7, \\
&\quad\; m0{,}2.m1{,}5.m2{,}10, m0{,}3.m2{,}3, \\
&\quad\; m0{,}3.m2{,}6, m0{,}3.m2{,}7, m0{,}3.m2{,}10\} \\
capture(m1) &= \{m1{,}5.m2{,}10, m1{,}2, m1{,}6\} \\
capture(m2) &= \{m2{,}8\}
\end{aligned}
$$

3.6.1. Memoria que escapa a un método

En virtud de caracterizar simbólicamente la cantidad de memoria que *escapa* a un método, utilizaremos el algoritmo descripto en Section 3.3, pero restringiremos la búsqueda a los puntos de creación que escapan al método:

$$\texttt{memEscapes}(m) = \texttt{computeAlloc}(m, escape(m))$$

Esta información puede ser utilizada para saber cuanta memoria deja reservada el método en las regiones activas (la región del llamador o sus regiones padres en la pila de llamados) después que su propia región es liberada, o para medir la cantidad de memoria que no puede ser recolectada por un recolector de basura luego de que el método termina.

En la tabla 3.4 mostramos las expresiones de consumo de memoria para los puntos de creación que escapan de $m1$. Observemos que las expresiones están definidas solamente en los parámetros del método.

3.6.2. Memoria capturada por un método

Para computar la expresión que sobre estima la cantidad de memoria reservada que es capturada por un método, utilizamos el algoritmo visto en 3.4, pero restringimos la búsqueda a puntos de creación que son capturados por el método:

memEscapes($m1$)=	$size(\texttt{Object[]}) \cdot k$	$m1$,3
	$+size(\texttt{Object[]}) \cdot \left(\frac{1}{2}k^2 + \frac{1}{2}k\right)$	$m1$,5.$m2$,3
	$+size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}k^3 + \frac{1}{2}k^2 + (per(k, [\frac{5}{6}, \frac{5}{6}, \frac{1}{6}])).k + per(k, [0, -\frac{4}{9}, -\frac{11}{9}])\right)$	$m1$,5.$m2$,6
	$+size(\texttt{Integer}) \cdot \left(\frac{1}{3}k^2 + \frac{2}{3}.k + per(k, [0, 0, \frac{1}{3}])\right)$	$m1$,5.$m2$,7

Cuadro 3.4: Polinomios para la cantidad de memoria que escapa de $m1$.

$$\texttt{memCaptured}(m) \texttt{ = computeAlloc}(m, \texttt{ } capture(m))$$

En la tabla podemos ver la cantidad de memoria capturada para nuestro ejemplo.

La expresión resultante es un estimador simbólico del tamaño de la región de memoria asociada al alcance del método. Esta información puede ser usada para especificar el tamaño de la región de memoria (en ambientes que utilicen algoritmos de reserva basados en "scope") que será reservada en tiempo de ejecución o para mejorar algoritmos de manejo de memoria.

memCaptured($m0$) = $size(\texttt{Ref0})$	$m0$,1
$+size(\texttt{Object[]}) \cdot mc$	$m0$,2.$m1$,3
$+size(\texttt{Object[]}) \cdot (\frac{1}{2}mc^2 + \frac{1}{2}mc) +$	$m0$,2.$m1$,5.$m2$,3
$+size(\texttt{Integer[]}) \cdot (\frac{1}{9}mc^3 + \frac{1}{2}mc^2 + per(mc, [-\frac{1}{6}, -\frac{1}{6}, -\frac{5}{6}])mc + per(mc, [0, -\frac{4}{9}, -\frac{11}{9}]))$	$m0$,2.$m1$,5.$m2$,6
$+size(\texttt{Integer}) \cdot (\frac{1}{3}mc^2 + \frac{2}{3}mc + per(mc, [0, 0, \frac{1}{3}])$	$m0$,2.$m1$,5.$m2$,7
$+size(\texttt{Integer[]}) \cdot mc$	$m0$,2.$m1$,5.$m2$,10
$+ size(\texttt{Object[]}) \cdot 2mc$	$m0$,3.$m2$,3
$+size(\texttt{Integer[]}) \cdot (\frac{4}{3}mc^2 + per(mc, [2, -\frac{2}{3}, \frac{2}{3}])mc + per(mc, [0, -\frac{2}{3}, -\frac{2}{3}]))$	$m0$,3.$m2$,6
$+size(\texttt{Integer}) \cdot (\frac{4}{3}mc + per(mc, [0, \frac{2}{3}, \frac{1}{3}])$	$m0$,3.$m2$,7
$+size(\texttt{Integer[]})$	$m0$,3.$m2$,10
$= size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{11}{6}mc^2 + (per(mc, [\frac{9}{2}, \frac{11}{6}, \frac{5}{2}]))mc + per(mc, [2, \frac{2}{9}, -\frac{2}{9}])\right) +$ $size(\texttt{Integer}) \cdot \left(\frac{1}{3}mc^2 + 2mc + per(mc, [0, \frac{2}{3}, \frac{2}{3}])\right) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{7}{2}mc\right) +$ $size(\texttt{Ref0})$	Total
memCaptured($m1$) = $size(\texttt{Ref0})$	$m1$,2
$+size(\texttt{Integer[]}) \cdot 9$	$m1$,6
$+size(\texttt{Integer[]}) \cdot k$	$m1$,5.$m2$,10
memCaptured($m2$) = $size(\texttt{Integer[]}) \cdot 4n$	$m2$,8

Cuadro 3.5: Expresiones simbólicas de la memoria capturada por los métodos $m0$, $m1$ y $m2$

3.7. Memoria necesaria para ejecutar un método

La técnica general no reconoce la presencia de un manejador de memoria que reclama los objetos no utilizados en tiempo de ejecución.

Particularmente, no toma en cuenta el hecho de que existe un manejador de memoria por "scope" que recolecta (garbage collect) las regiones no utilizadas, en cuyo caso arrojará una cota superior excesivamente pesimista.

Debemos notar que, en este esquema, aunque un método podría ser invocado varias veces, poseerá como máximo una región activa por cada método y su tamaño cambiará de acuerdo al contexto de llamada (el valor asignado a sus parámetros cada vez que es invocado).

Consideremos el método $m0$ en el ejemplo de la Figura 3.1. En la ubicación $m0$,2, $m0$ llama a $m1$ el cual llama a $m2$. En la ubicación $m0$,3, $m0$ llama a $m2$. Bajo un modelo de memoria por scope, donde a cada método se le asigna una región, habrá tres regiones activas para la cadena de llamados $m0 \rightarrow m1 \rightarrow m2$, y dos regiones activas para la cadena de llamados $m0 \rightarrow m2$. Estas dos cadenas son independientes debido a que no puede estar simultáneamente activas.

Podemos observar esta situación en la Figura 3.3. A través del eje X pasa el flujo de la ejecución. En primer lugar, se van reservando las regiones correspondientes a la primera cadena y cuando termina la ejecución de $m2$ en la primer cadena, se liberan estas regiones, dejando solamente cargado $m0$. Luego, se comienzan a cargar las regiones de la segunda cadena, de manera totalmente independiente de lo anterior.

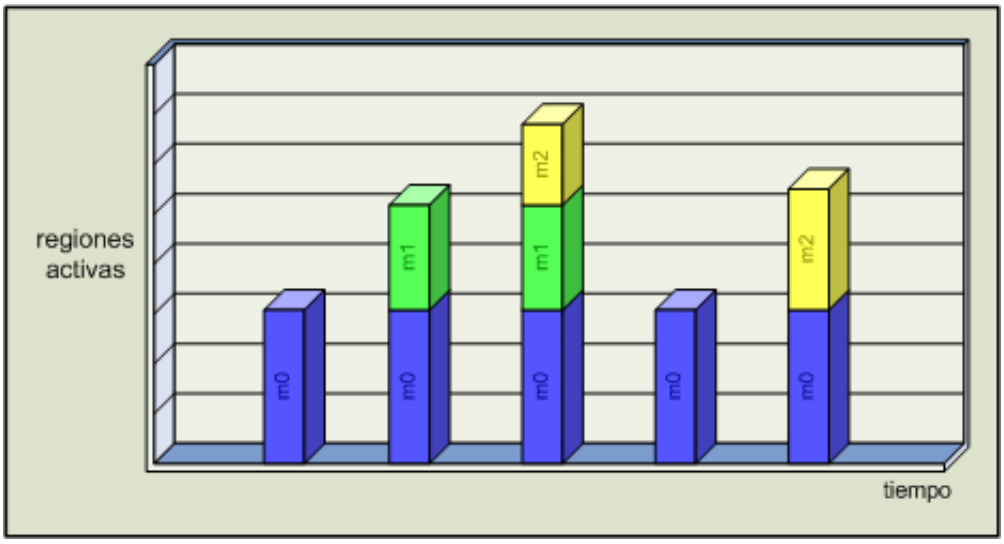


Figura 3.3: Regiones activas para una ejecución

En este nuevo contexto, la memoria necesaria para ejecutar un método es relativa

no sólo al conteo correspondiente sino también a la pila de llamadas que se ha realizado dentro del grafo de llamados. De esta manera, dada una cadena de llamados es posible estimar el tamaño de su región asociada computando el máximo tamaño para cada región considerando el contexto de llamada.

El planteo y formulación de este modelo constituye un interesante desafío que constituye uno de los aportes de esta tesis. Dicho modelo requiere, como ya hemos mencionado en varias oportunidades, la resolución de un complejo problema de optimización simbólico. Para ello hemos evaluado varias alternativas (que describiremos más adelante) siendo la más viable la aplicación de la técnica que describiremos a continuación (y aplicaremos luego).

Capítulo 4

Bases de Bernstein

En este capítulo introduciremos las nociones necesarias para presentar el método que hemos elegido como aproximación a la resolución del problema de optimización descripto en el capítulo anterior.

Las bases de Bernstein son un método general para obtener cotas de polinomios. Avances recientes descriptos en [CT04] permiten ser aplicadas sobre dominios de poliedros, lo cual representa una interesante línea de resolución para nuestro problema.

Comenzaremos mencionando los conceptos básicos de las bases de Bernstein junto con algunos detalles sobre su cálculo, y luego avanzaremos sobre el aporte realizado en [CT04] para la utilización paramétrica de las bases y finalmente, la introducción de los poliedros.

4.1. Conceptos preliminares

Los polinomios de Bernstein son polinomios particulares que forman una base para el espacio de polinomios. Esto es debido a que cualquier polinomio puede ser expresado en esta base a través de coeficientes , llamados los coeficientes de Bernstein. Estos coeficientes poseen interesantes propiedades y pueden ser calculados mediante una formula cerrada.

Dada la propiedad de la cápsula convexa de Bernstein [Far93], los valores de las especializaciones del polinomio estan acotados por los valores del mínimo y máximo coeficiente de Bernstein. La fórmula cerrada permite el cálculo simbólico de estos coeficientes de Bernstein dando un interés suplementario al uso de esta teoría. Otra principal consecuencia es que los cálculos involucrados tienen baja complejidad.

4.1.1. Polinomios y representaciones

Existen diversos usos para los polinomios en el campo de las ciencias de la computación. La representación más simple de un polinomio es su forma implícita en potencias.

Un polinomio de grado $n \in \mathbb{N}$ en la variable x se define como:

$$p(x) = \sum_{k=0}^{n} a_k x^k$$

con $a_k \in \mathbb{R}$.

Otra posible representación de los polinomios es en términos de las bases de Bernstein. La base de Bernstein tiene, desde el punto de vista geométrico, interesantes ventajas.

Pasaremos a definir esta representación y destacar algunas propiedades generales .

Definición

Dado $n \in \mathbb{N}$ las funciones de Bernstein de grado n en el intervalo unitario $[0, 1]$ están definidas por:

$$B_k^n(x) = \binom{n}{k} x^k (1-x)^{n-k}, \forall x \in [0,1], k = 0, 1, ..., n.$$

Las funciones de Bernstein $B_k^n(x)$ son linealmente independientes y forman una base del espacio de polinomios de grado n.

A veces es más conveniente considerar el intervalo unitario [0,1] como el área de interés. Sin embargo, esto no es una restricción real ya que la generalización a un intervalo arbitrario es simple. En el intervalo $[x', x'']$ con $x', x'' \in \mathbb{R}$ la base de Bernstein de grado $n \in \mathbb{N}$ se define como:

$$B_k^n(x) = \binom{n}{k} \frac{x - x'^k (x'' - x)^{n-k}}{(x'' - x')^n}, \forall x \in [0,1], k = 0, 1, ..., n.$$

Un polinomio arbitrario puede ser expresado en términos de la base de Bernstein de grado n. Un polinomio en forma de Bernstein de grado n tiene la siguiente forma:

$$p(x) = \sum_{k=0}^{n} m_k^n B_k^n(x)$$

donde m_k^n son los coeficientes de Bernstein correspondientes a la base de grado n.

Comparada con la forma de potencias, la forma de Bernstein tiene algunas propiedades geométricas especiales, algunas de las cuales serán mencionadas a continuación.

Propiedades

En esta sección daremos solamente un resumen de algunas propiedades de los polinomios de Bernstein. Más detalles se pueden encontrar en el trabajo de Bowyer y Woodwark [BW93] y Farouki y Rajan [FR87, FR88].

1. Los polinomios de Bernstein son invariantes con respecto a transformaciones afines.

2. Es posible realizar una generación recursiva de la base de orden n desde la base de orden $n-1$. Para la base de Bernstein en el intervalo unitario $[0,1]$ la recursión está dada por:

$$B_k^n = (1-x)B_k^{n-1}(x) + xB_k^{n-1} - 1(x), k = 0, 1, \cdots, n.$$

3. Todos los términos de la base de Bernstein son positivos en el intervalo donde están definidos y su suma es igual a 1:

$$B_k^n \geq 0, k = 0, 1, \cdots, n \text{ y } \sum_{k=0}^{n} B_k^n(x) = 1.$$

   Farouki y Rajan [FR87] muestran que este hecho da una cota para el polinomio $p(x)$:

$$\min_{0 \leq k \leq n} m_k^n \leq p(x) \leq \max_{0 \leq k \leq n} m_k^n$$

   Una cota más ajustada es dada por la cápsula convexa determinada por los coeficientes de Bernstein. En dos dimensiones la cápsula convexa es dada por un polígono; en tres dimensiones es representada por un poliedro convexo.

4. Un polinomio $p(x)$ de grado n puede ser representado en términos de la base de Bernstein de grado $n+1$ a través de un procedimiento conocido como elevación de grado. Si m_k^n son los coeficientes de Bernstein de la base de grado n, los coeficientes m_k^{n+1} en la siguiente base (en orden) estarán dados por:

$$m_k^{n+1} = \omega_k m_{k+1}^n + (1-\omega_k)m_k^n, \text{ donde } \omega_k = \frac{k}{n+1}$$

   para $k = 1, 2, \cdots, n$ y $m_0^{n+1} = m_0^n, m_{n+1}^{n+1} \leq m_n^n$.

5. Farouki y Rajan [FR87, FG96] muestran que un polinomio en forma de Bernstein siempre está mejor condicionado que un polinomio en la forma de potencias para la determinación de la raíces simples en el intervalo [0,1]. Además para raíces en un intervalo arbitrario [a,b], el número de condición de la raíz es más pequeño en la base de Bernstein.

6. La base de Bernstein tiene una mejor estabilidad numérica que la forma de potencias. Spencer en [Spe94] dio la siguiente definición: *La estabilidad numérica es una propiedad de un algoritmo que mide cuan propenso es a generar y propagar errores de redondeo y errores inherentes en los datos de entrada.* Sin embargo, si la conversión entre la dos formas es hecha frecuentemente, inestabilidades numéricas pueden ser reintroducidas y la propiedad es obviamente perdida. Una forma de medir esta propiedad es perturbar los coeficientes de ambas representaciones del mismo polinomio. El polinomio en forma de Bernstein tendrá una menor cota de error en un punto que la cota de error generada en la forma de potencias, a menudo por varios órdenes de magnitud.

4.2. Conversión a la base

Dado que la representación más común de los polinomios es en la base de potencias, es necesario tener bien claros los procedimientos de conversión a la base de Bernstein.

El proceso es similar tanto para el caso de una variable como de múltiples variables, pero en este último se vuelve más complicado de seguir dada la gran cantidad de notación involucrada. Por lo tanto, presentaremos en primer lugar el cálculo para el caso univariable para dar claridad al proceso y luego pasaremos el de múltiples variables.

Finalmente pasaremos a mostrar con un nivel de detalle un poco más profundo la transformación de Bernstein sobre dominios dados por restricciones lineales, que es la versión que nos será útil en adelante (y que hemos implementado).

4.2.1. Univariable

En esta sección mostraremos la conversión a la base de polinomios con una única variable sobre distintos dominios.

En el intervalo unitario [0,1]

Sea $p(x)$ un polinomio de grado $n \in \mathbb{N}$. Sus formas equivalentes de potencias y de Bernstein son:

$$p(x) = \sum_{k=0}^{n} a_k x^k = \sum_{k=0}^{n} m_k^n B_k^n(x).$$

Cada conjunto de coeficientes (a_k o m_k^n respectivamente) puede ser computado a partir del otro. Por ejemplo:

$$a_k = \sum_{j=0}^{k} (-1)^{(k-j)} \binom{n}{k} \binom{k}{j} m_j^n$$

$$m_k^n = \sum_{j=0}^{k} \frac{\binom{k}{j}}{\binom{n}{j}} a_j.$$

La fórmula anterior muestra la forma de conversión de un polinomio de una sola variable desde su forma de potencias en la forma de Bernstein. Veamos un ejemplo:

$$p(x) = x^3 - 5x^2 + 2x + 4 = 4B_0^3 + \frac{14}{3}B_1^3 + \frac{11}{3}B_2^3 + 2B_3^3$$

donde $B_0^3 = (1-x)^3, B_1^3 = 3x(1-x)^2, B_2^3 = 3x^2(1-x)$ y $B_3^3 = x^3$.

Es simple ver que esta misma operación puede ser expresada en términos matriciales.

En un intervalo arbitrario [a,b]

La restricción $x \in [0,1]$ puede ser eliminada extendiendo el dominio de los polinomios de Bernstein a $[a,b]$:

$$B_k^n(x) = \binom{n}{k} \left(\frac{x-a}{b-a}\right)^k \left(1 - \frac{x-a}{b-a}\right)^{n-k}, \forall x \in [a,b]$$

4.2.2. Multivariable

Ahora describiremos la versión de múltiples variables de las bases de Bernstein.

Sea

$C_N^K = C_{n_1}^{k_1} \cdot \cdots \cdot C_{n_l}^{k_l}$, donde $\bar{0} \leq K = (k_1, .., k_l) \leq N = (n_1, ..., n_l)$, $(K, N) \in \mathbb{N}_0^l$,

y $C_n^k = \frac{n!}{k!(n-k)!}, 0 \leq k \leq n$, $(k, n) \in \mathbb{N}_0$

$I - J = (i_1 - j_1, \ldots, i_l - j_l)$, donde $I = (i_1, \ldots, i_l)$, $J = (j_1, \ldots, j_l)$, $I, J \in \mathbb{N}_0^l$
$\alpha^I = \alpha_1^{i_1} \cdot \cdots \cdot \alpha_l^{i_l}$, donde $\alpha = (\alpha_1, \ldots, \alpha_l) \in \mathbb{R}^l$

Para polinomios con coeficientes constantes:

$$p(\bar{x}) = \sum_{\bar{0} \leq I \leq N} a_I \cdot \bar{x}^I \in \mathbb{R}[\bar{x}], \quad \bar{x} \in D \subset \mathbb{R}^l, a_I \in \mathbb{R}$$

y con máximo multi-grado N, los polinomios de Bernstein y los coeficientes de Bernstein se definen de la siguiente manera:

El I-ésimo polinomio de Bernstein en la caja unitaria $[0,1]^l$, donde $I = (i_1, \ldots, i_l)$ y $N = (n_1, \ldots, n_l)$ se define como:

$$B_{N,I}^{[0,1]^l}(\bar{x}) = b_{n_1,i_1}^{[0,1]}(x_1) \cdot \cdots \cdot b_{n_l,i_l}^{[0,1]}(x_l), \bar{0} \leq I \leq N, \bar{x} = (x_1, \ldots, x_l) \in \mathbb{R}^l$$

Los polinomios de Bernstein se definen en el espacio l dimensional completo $\mathbb{R}^l$ y definen una base para los polinomios $p(\bar{x})$:

$$p(\bar{x}) = \sum_{\bar{0} \leq I \leq N} b_I^{[0,1]^l} B_{N,I}^{[0,1]^l}(\bar{x}) = \sum_{\bar{0} \leq I \leq N} a_I \cdot \bar{x}^I,$$

donde los coeficientes $b_I^{[0,1]^l}$ son los coeficientes de Bernstein para el polinomio $p(\bar{x})$ en la caja unitaria $[0,1]^l$. Estos son determinados por:

$$b_I^{[0,1]^l} = \sum_{\bar{0} \leq J \leq I} \frac{C_I^J}{C_N^J} a_J, \quad \bar{0} \leq I \leq N.$$

Teorema Sea $p(\bar{x})$ un polinomio de multi-grado N en la caja unitaria $[0,1]^l$. Para todo $\bar{x} \in [0,1]^l$:

$$\min_{\bar{0} \leq I \leq N} b_I^{[0,1]^l} \leq p(\bar{x}) \leq \max_{\bar{0} \leq I \leq N} b_I^{[0,1]^l}$$

La demostración de este teorema no será explicada aquí ya que excede el alcance de este trabajo (aunque pueden verse en las referencias (demostraciones similares para los casos más simples) en 4.1.1).

4.2.3. Conversión para dominios sobre poliedros

Hasta aquí hemos visto varias conversiones para distintos casos crecientes en complejidad para ilustrar el uso básico de esta base y sus técnicas. Claramente, el problema que nos ocupa no es modelado por los casos vistos.

En [CT04] se avanza sobre los distintos tipos de casos soportados, agregando parámetros simbólicos relacionados a la entrada del problema en los intervalos y complicando un poco los dominios dados por las cajas. Ninguna de estos avances nos permite resolver nuestro problema.

Recordando el problema a resolver, es decir, el cómputo de los requerimientos de memoria notamos que las variables de los polinomios (estimación de consumo de memoria) se mueven por su dominio restringidas por restricciones lineales (los invariantes). Por lo tanto sería ideal tener una extensión de ésta técnica que nos permita definir un dominio de las variables a partir de restricciones lineales, esto es, un poliedro convexo, ya que las distintas versiones con cajas no nos sirven.

Afortunadamente, Philippe Clauss continuó expandiendo su trabajo y obtuvo una técnica para este tipo de dominios, ya que existen múltiples aplicaciones en el análisis estático y optimización de código que utilizan invariantes u otros predicados lineales para las cuales serían muy útil poseer la capacidad de optimizar polinomios.

En adelante, describiremos el trabajo que finalmente da la solución teórica al problema de estimación de memoria.

Representación de poliedros

Es un hecho conocido que cualquier poliedro convexo puede ser definido como un conjunto de todas las combinaciones convexas de sus vértices: sean $v_1, v_2, ..., v_{nv}$ los vértices de P. Entonces cualquier punto x en P puede ser definido como una combinación lineal de los vértices, llamada una *combinación convexa*, $x = a_1v_1 + a_2v_2 + ... + a_{nv}v_{nv}$ donde $\forall\ a_i, i = 1..nv, 0 \leq a_i \leq 1$ y $\sum_{i=1}^{nv} a_i = 1$.

Cuando consideramos un poliedro convexo paramétrico, la misma definición de sus puntos puede ser usada. Sin embargo, el poliedro convexo tiene primeramente que ser descompuesto en poliedros de figuras constantes definidos sobre dominios de los parámetros convexos y adyacentes (ya que sino sería imposible operar debido a que para distintos valores de los parametros podrían cambiar ciertas propiedades de la figura). Para dichos dominios de los parámetros, las coordenadas de los vértices y el número de vértices son constantes. Esto será tenido en cuenta al iniciar el cálculo, ya que necesitaremos los vértices para generar las combinaciones convexas.

Teniendo en cuenta la representación de los poliedros veremos ahora como aplica la transformación de Bernstein sobre este nuevo dominio.

Sobre la expansión de Bernstein

Los polinomios de Bernstein constituyen una base para los polinomios de un grado dado y un cierto número de variables. Pueden ser vistos desde un punto de vista geométrico como los coeficientes $a_1, a_2, ..., a_{nv}$ de una combinación convexa, dado que sus valores están entre 0 y 1 y su suma es 1. Por lo tanto los coeficientes de Bernstein pueden ser vistos como los vértices de todos los posibles valores del polinomio. Sin embargo, todas las combinaciones convexas de estos vértices no corresponden a valores efectivamente tomados por el polinomio. Esto es el por qué de que ésta forma específica de los polinomios de Bernstein introduce restricciones en sus posibles valores, imponiendo condiciones sobre los valores (o la suma) de los coeficientes. De cualquier manera, dado que los coeficientes de Bernstein puede ser vistos como vértices, naturalmente proveen cotas superiores e inferiores de los valores del polinomio.

Consideremos ahora un polinomio cuyas variables están definidas sobre un poliedro convexo P. El vector de las variables x puede entonces ser definido como una combinación

convexa de los vértices de P, $v_1, v_2, ..., v_{nv}$: $x = a_1v_1 + a_2v_2 + ... + a_{nv}v_{nv}$ donde $\forall\ a_i, i = 1..nv, 0 \leq a_i \leq 1$ y $\sum_{i=1}^{nv} a_i = 1$.

Luego x puede ser reemplazado en el polinomio con su forma expandida $a_1v_1 + a_2v_2 + ... + a_{nv}v_{nv}$. De esta manera, obtenemos un nuevo polinomio, cuyas variables son $a_1, a_2, ..., a_{nv}$ definidas sobre la caja unitaria $[0,1]^{nv}$. Algunas cotas inferiores y superiores de este polinomio podrían ser obtenidas calculando sus coeficientes de Bernstein. Sin embargo, estos cotas no serían muy ajustadas, dado que la propiedad que dice que $\sum_{i=1}^{nv} a_i = 1$ y que relaciona las variables no es tomada en cuenta. Por ejemplo, el caso donde todas las variables son iguales a 1 será considerado de manera abusiva.

Analizando en profundidad la base de Bernstein, es posible observar que los polinomios de Bernstein son todos los monomios obtenidos de la expansión de $(x + (1 - x))^n$, donde n es el grado del polinomio. Estos monomios son tales que:

1. Su suma es 1
2. Cada monomio varía entre 0 y 1
3. Todos los monomios constituyen una base para los polinomios considerados para un grado dado y un número de variables.

Puede verse que cualquier conjunto de polinomios con esas 3 propiedades constituye una base cuyos coeficientes dan cotas superiores e inferiores de los valores del polinomio.

Veamos de nuevo un polinomio cuyas variables son coeficentes de una combinación convexa. Para dicho polinomio, una primera idea sería constriur una base a partir de la expansión de $(a_1 + a_2 + ... + a_v)^n$. Sin embargo, los monomios que se obtienen no parecen constituir una base. Por ejemplo, los términos constantes no pueden ser definidos en esta base.

La propiedad que señala que $\sum_{i=1}^{nv} a_i = 1$ permite la reescritura del polinomio como otro polinomio cuyos monomios son todos de grado n. Dicho polinomio puede entonces ser generado a partir de la base construída con $(a_1 + a_2 + ... + a_v)^n$. La transformación se realiza de la siguiente manera: cualquier monomio M de grado $i < n$ puede ser reescrito como una suma de monomios de grado n expandiendo $M = M\,(a_1 + a_2 + ... + a_v)^{n-i}$. Dado que $a_1 + a_2 + ... + a_v$ es igual a 1, la igualdad vale.

Veamos entonces, un ejemplo de aplicación de este nuevo enfoque.

Ejemplo

Consideremos el polinomio $\frac{1}{2}i^2 + \frac{1}{2}i + j$ sobre la caja $[0, N] \times [0, i]$. Esta caja define un poliedro convexo cuyos vértices son:

$\begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}N\\0\end{pmatrix}$ y $\begin{pmatrix}N\\N\end{pmatrix}$. Por lo tanto, cualquier punto $\begin{pmatrix}i\\j\end{pmatrix}$ es una combinación convexa de los vértices: $\begin{pmatrix}i\\j\end{pmatrix} = a_1\begin{pmatrix}0\\0\end{pmatrix} + a_2\begin{pmatrix}N\\0\end{pmatrix} + a_3\begin{pmatrix}N\\N\end{pmatrix}, 0 \leq a_i \leq 1, \sum_{i=1}^{3} a_i = 1.$

Reemplazando $\binom{i}{j}$ con su combinación convexa, obtenemos un nuevo polinomio cuyas variables son a_1, a_2, a_3:

$$\frac{1}{2}N^2a_2^2 + N^2a_2a_3 + \frac{1}{2}N^2a_3^2 + \frac{1}{2}Na_2 + \frac{3}{2}Na_3$$

Los monomios que tiene grados menores que 2 son transformados en sumas de monomios de grado 2:

- $\frac{1}{2}Na_2 = \frac{1}{2}Na_2(a_1 + a_2 + a_3)$
- $\frac{3}{2}Na_3 = \frac{3}{2}Na_3(a_1 + a_2 + a_3)$

El polinomio final es:

$$(\frac{1}{2}N^2 + \frac{1}{2}N)a_2^2 + (\frac{1}{2}N^2 + \frac{3}{2}N)a_3^2 + \frac{1}{2}Na_1a_2 + \frac{3}{2}Na_1a_3 + (N^2 + 2N)a_2a_3$$

La base es construída a partir de la expansión de $(a_1 + a_2 + a_3)^2$ dando los siguientes monomios:

$$\begin{aligned} P1 &= a_1^2 \\ P2 &= a_2^2 \\ P3 &= a_3^2 \\ P4 &= 2a_1a_2 \\ P5 &= 2a_1a_3 \\ P6 &= 2a_2a_3 \end{aligned}$$

Los coeficienes de estos polinomios son utilizados en la matriz T permitiendo cambiar la base:

$$\{a_1^2, a_2^2, a_3^2, a_1a_2, a_1a_3, a_2a_3\}$$

hacia la base:

$$\{P1, P2, P3, P4, P5, P6\}$$

Sea B el vector columna de los monomios que definen la nueva base y sea C el vector fila de los coeficientes en el polinomio. El polinomio en la nueva base es obtenido calculando $(C \cdot T^{-1} \cdot B)$:

$$0P1 + (\frac{1}{2}N^2 + \frac{1}{2}N)P2 + (\frac{1}{2}N^2 + \frac{3}{2}N)P3 + \frac{1}{4}NP4 + \frac{3}{4}NP5 + (\frac{1}{2}N^2 + N)P6$$

Podemos concluir, de acuerdo a las propiedades vistas, que el polinomio varía entre 0 y $\frac{1}{2}N^2 + \frac{3}{2}N$. Es posible ver que estas cotas son exactas.

Aplicación al problema

Como hemos visto en el capítulo anterior, la resolución del problema que nos ocupa plantea un desafío interesante: resolver un problema de optimización no-lineal (de un polinomio o pseudo-polinomio) cumpliendo restricciones lineales, de manera simbólica para que su resultado pueda quedar expresado en términos de los parámetros y ser finalmente evaluado en tiempo de ejecución.

Todo indicaría que la técnica de Bernstein cumple la mayor parte de estos requisitos. En los siguientes capítulos enfrentaremos la resolución viendo como se ajusta la técnica a nuestro problema.

Capítulo 5

Resolviendo el problema de optimización

Dados los detalles de las técnicas para la estimación de memoria y presentada la técnica de Bases de Bernstein para la búsqueda de cotas, pasaremos ahora a comenzar a realizar una correspondencia entre las técnicas de optimización y su aplicación al problema de optimización.

En primer lugar, definiremos el modelo considerado, luego comentaremos algunas alternativas para la resolución, introduciremos la integración con un pequeño y simple ejemplo y finalmente veremos los detalles del cálculo en vistas a la implementación.

5.1. Modelo para el cálculo de memoria requerida

Con el objetivo de obtener la resolución de la estimación de memoria requerida para ejecutar un método, plantearemos ahora el modelo definido para representar la ejecución de métodos considerando el grafo de llamados y los invariantes correspondientes a la pila de llamados.

Dada una cadena de llamados podemos estimar el tamaño de su región asociada computando el máximo tamaño para cada región considerando el contexto de llamada.

Asumamos un manejo de memoria por "scope" donde existe una región asociada a cada método. Sea $\texttt{rSize}_{m_r}^{\pi.m}$ una función la cual arroja el tamaño de la mayor región creada por cualquier llamado a m con una pila de llamados π en un programa comenzando por el método m_r.

Supongamos que somos capaces de computar **rsize** para cada método y para cada cadena de llamados. Para computar la cantidad de memoria requerida para correr el método $m0$ podemos hacer lo siguiente:

$$
\begin{aligned}
\texttt{memRq}_{m0}^{m0} &= \texttt{rSize}_{m0}^{m0} + max\Big(\texttt{memRq}_{m0}^{m0,2.m1}, \texttt{memRq}_{m0}^{m0,3.m2}\Big)\\
\texttt{memRq}_{m0}^{m0,2.m1} &= \texttt{rSize}_{m0}^{m0,2.m1} + \texttt{memRq}_{m0}^{m0,2.m1,5.m2}\\
\texttt{memRq}_{m0}^{m0,2.m1,5.m2} &= \texttt{rSize}_{m0}^{m0,2.m1,5.m2}\\
\texttt{memRq}_{m0}^{m0,3.m2} &= \texttt{rSize}_{m0}^{m0,3.m2}
\end{aligned}
$$

Estas expresiones pueden ser reducidas a:

$$
\begin{aligned}
\texttt{memRq}_{m0}^{m0} &= \texttt{rSize}_{m0}^{m0} +\\
&\quad max\Big(\texttt{rSize}_{m0}^{m0,2.m1} + \texttt{rSize}_{m0}^{m0,2.m1,5.m2}, \texttt{rSize}_{m0}^{m0,3.m2}\Big)
\end{aligned}
$$

En general esta función puede ser definida de la siguiente manera:

$$
\texttt{memRq}_{m_r}^{\pi.m} = \texttt{rSize}_{m_r}^{\pi.m} + \max_{(m,l,m_i)\ \in\ edges(CG_{m_r}\downarrow\pi.m)} \texttt{memRq}_{m_r}^{\pi.m.l.m_i}
$$

donde $CG_{m_r} \downarrow \pi.m$ es una proyección sobre el camino $\pi.m$ del grafo de llamados del método m_r y $edges$ es el conjunto de arcos del grafo.

Ahora, nos centraremos en la manera de computar la función **rsize**.

En el ejemplo, el método $m0$ llama al método $m1$ el cual llama k veces al método $m2$. En cada invocación, el tamaño de la región para $m2$ cambia debido a que el número de objetos creados depende del parámetro n. Entonces, la expresión para $\texttt{rSize}_{m0}^{m0,2.m1,5.m2}$ debe ser el tamaño máximo de la región para el método $m2$ entre todos los posibles k.

En virtud de obtener dicha expresión, es necesario elegir un valor válido para n (que satisfaga el invariante para el estado abstracto m0.2.m1.5.m2, esto es el punto de entrada del método $m2$ considerando la pila de llamados m0.2.m1.5) que maximice la expresión que caracteriza la cantidad de memoria capturada por el método $m2$.

Dado que la región para $m2$ está definida por la expresión $size(Int) \cdot 4n$ (ver tabla 3.5), la región más grande es producida cuando $m2$ es llamada con $n = i = k = mc$.

Por lo tanto, $\texttt{rSize}_{m0}^{m0,2.m1,5.m2} = size(Int) \cdot 4mc$.

$\texttt{rSize}_{m_r}^{\pi.m}$ debe ser una expresión (en términos de los parámetros de m_r) que represente el tamaño de la región más grande creada por alguna llamada a m con una pila de llamados π considerando un programa comenzando con m_r.

Puede ser definida de la siguiente manera:

$$
\begin{aligned}
\texttt{rSize}_{m_r}^{\pi.m} = \lambda \vec{p_{m_r}} \bullet\ &(\text{Maximizar memCaptured}(m)\\
&\text{sujeto a } \mathcal{I}_{\pi}^{m_r}[P_{m_r}/\vec{p_{m_r}}])
\end{aligned}
$$

Debemos notar que I^{m_r} tiene las siguientes variables libres: P_m (los parámetros del método m), P_{m_r} (los parámetros del método m_r) y W (otras variables en el invariante)

mientras que `memCaptured(m)`, la expresión simbólica para la memoria capturada por m, está solamente expresado en términos de P_m.

En el ejemplo, para computar

$$\begin{aligned}
&\texttt{rSize}_{m0}^{m0,2.m1,5.m2} = \\
&= \lambda\overline{mc}\,.\,\max(size(Int)\cdot 4n) \text{ s.t.}\{k = mc, 1 \leq i \leq k, n = i\}[mc/\overline{mc}] \\
&= \lambda\overline{mc}\,.\,\max(size(Int)\cdot 4n) \text{ s.t.}\{k = \overline{mc}, 1 \leq i \leq k, n = i\} \\
&= \lambda\overline{mc}\,.\,\max(size(Int)\cdot 4n) \text{ s.t.}\{1 \leq n \leq \overline{mc}\} \\
&= \lambda\overline{mc}\,.\,size(Int)\cdot 4\overline{mc}
\end{aligned}$$

Para calcular $\texttt{rSize}_{m0}^{m0}$, no es necesaria ninguna maximización. Esto se debe a que `memCaptured(m)` (ver tabla 3.5) está expresada únicamente en términos de los parámetros del método $m0$ y del invariante global.

De acuerdo a este análisis, la tabla 5.1 muestra las expresiones resultantes para $\texttt{rSize}_{m0}\pi.m$.

m	$\texttt{rSize}_{m0}^{\pi.m}$
$m0$	$size(Int) \cdot \left(\frac{1}{9}mc^3 + \frac{13}{6}mc^2 + (2 + per(mc, [\frac{11}{6}, -\frac{5}{6}, -\frac{1}{6}]))mc + per(mc, [0, -\frac{4}{9}, -\frac{11}{9}]) + 2\right) + size(Object) \cdot \left(\frac{1}{2}mc^2 + \frac{7}{2}mc\right) + size(RefO)$
$m0$,$2.m1$	$size(Int)\cdot 9 + size(RefO)$
$m0$,$2.m1$,$5m2$	$size(Int)\cdot 4mc$
$m0$,$3.m2$	$size(Int)\cdot 8mc$

Cuadro 5.1: Expresiones para la función `rSize` para el ejemplo

La expresión simbólica $\texttt{memRq}_{m0}^{m0}$ puede ser reducida a:

$$\begin{aligned}
\texttt{memRq}_{m0}^{m0} \quad = \quad & size(\texttt{Integer[]}) \cdot \Big(\frac{1}{9}mc^3 + \frac{11}{6}mc^2 \\
& + (4 + per(mc, [\frac{9}{2}, \frac{11}{6}, \frac{5}{2}]))mc \\
& + per(mc, [2, \frac{2}{9}, -\frac{2}{9}])\Big) + \left\{ \begin{array}{ll} (9 + mc) & \text{if } mc \leq 3 \\ 4mc & \text{if } mc > 3 \end{array} \right) \\
& + size(\texttt{Integer}) \cdot \Big(\frac{1}{3}mc^2 + 2mc + per(mc, [0, \frac{2}{3}, \frac{2}{3}])\Big) \\
& + size(\texttt{Object[]}) \cdot \Big(\frac{1}{2}mc^2 + \frac{7}{2}mc\Big) + 2 \cdot size(\texttt{Ref0})
\end{aligned}$$

La resolución simbólica de este problema de maximización constituye un interesante problema que nos planteamos resolver en esta tesis.

El carácter simbólico de la resolución, permitiría evitar computaciones excesivamente caras en tiempo de ejecución, lo cual es un requerimiento fundamental para su aplicación en entornos embebidos o de tiempo real.

A continuación presentamos algunas ideas útiles para discutir sobre la correctitud del modelo propuesto. Lo que nos interesa mostrar que con el mismo nunca se obtendra una subaproxicación del valor real, siempre se obtendrá un valor mayor o igual. Como punto de partida, asumamos que poseemos un cálculo sobre-aproximado de la función **rsize** para cada una de las cadenas en cuestión. Esto es así debido a que:

- Los invariantes que se encuentran para realizar la estimación por cada método siempre son más generales, por lo tanto, siempre se obtiene un conteo de puntos más grande (o igual) que el real.
- El método (que más adelante describiremos) utilizado para resolver las máximizaciones o bien es exacto, o bien sobreaproxima las cotas.

Veamos entonces, por qué el cálculo de la función **memRq** se corresponde con una sobre aproximación de la memoria necesaria para ejecutar un método.

En primer lugar, analizemos que es lo que pasa cuando se ejecuta un cierto método $m0$ con las regiones de memoria generadas. Teniendo en cuenta el modelo de memoria por "scope", sabemos que cada método que se ejecuta genera una región de memoria asociada, donde viven los objetos que están dentro del tiempo de vida del método. Es decir, todos los objetos creados durante la ejecución del método $m0$ y que no escapan a él. Esta región de memoria es la que está considerada en el **rsize** correspondiente a $m0$.

Lo mismo ocurrirá con los métodos que sean llamados desde éste método raíz. Cada uno de ellos generará una región de memoria, correctamente estimada por los **rsize** correspondientes (considerando la cadena de llamados desde $m0$ y todos sus invariantes intermedios para el cálculo).

Lo que nos queda por ver, es la correctitud de la expresión de **memRq** globalmente. Para ello analizemos que pasa cuando se realizan llamados a métodos desde $m0$. Básicamente, para cada método llamado se genera una región de memoria intermedia, y en el último método de la cadena será donde se realiza la captura de memoria como ya hemos comentado. Por lo tanto, en principio lo que hay que estimar para cada uno de estos llamados es cada una de las secciones de las cadenas, tomando como hoja cada uno de los métodos incluídos. Esto es lo que realizamos cuando hacemos la suma en la expresión de **memRq** dentro de una cadena, ya que todas estas regiones estarán *simultáneamente* activas.

Finalmente, si en el método $m0$ (y así recursivamente) se realizan llamados a varios métodos éstos serán independientes en cuanto a la ocupación de memoria ya que al ser un

llamado secuencial la primera cadena se descargará de memoria antes de que se ejecute la siguiente. Por eso es que se calcula el máximo entre estas alternativas.

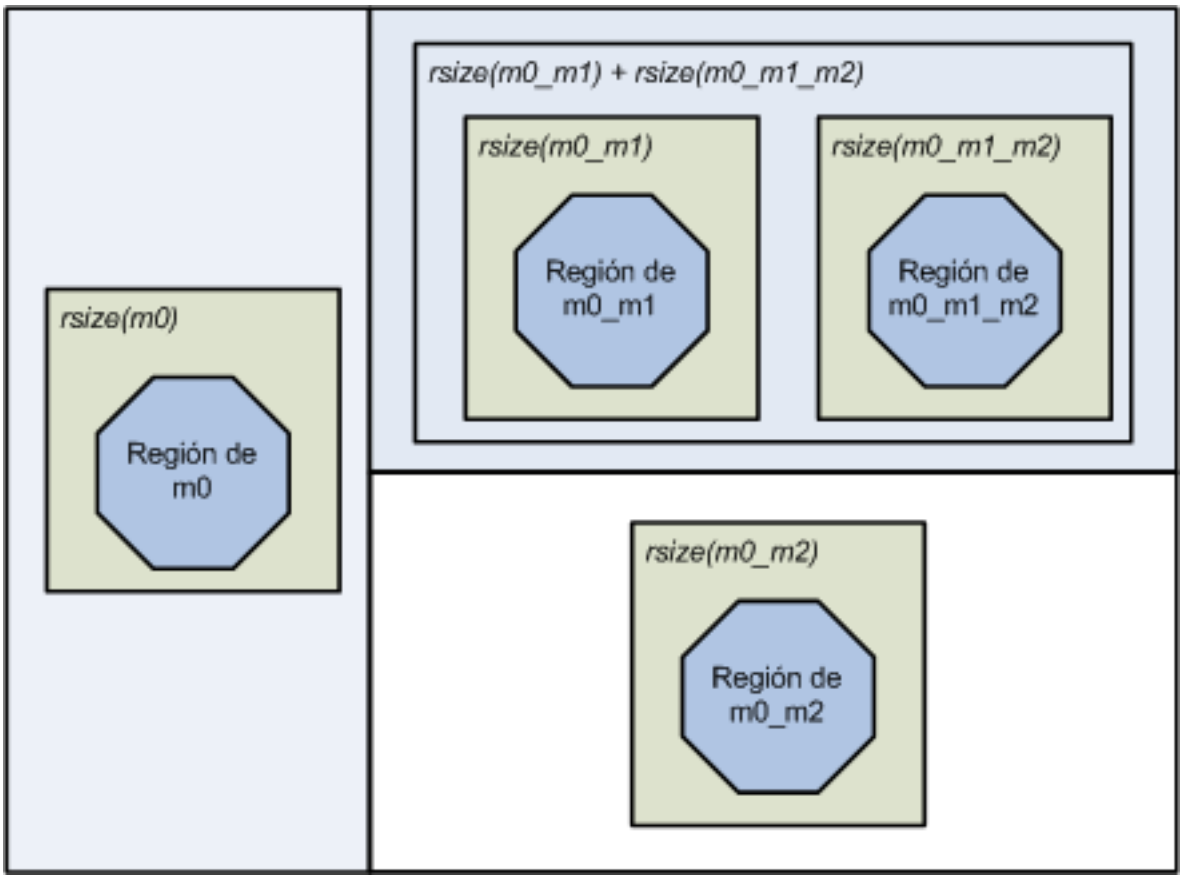


Figura 5.1: Esquema de cálculo para un fragmento de código de ejemplo

En la Figura 5.1 podemos observar un esquema de ésta situación. Se pueden ver los distintos valores de las regiones de memoria, (sobre) aproximadas por los **rsize** correspondientes. En el cuadro superior aparece la suma de dos cadenas que simúltaneamente se encuentran activas (ya que son secciones de la misma) y en el cuadro inferior podemos ver la otra cadena que no es simúltanea con lo anterior. Finalmente a la izquierda, se observa el cálculo para $m0$ que es la raíz. La región máxima corresponde a los cuadros sombreados (izquierdo y superior).

En conclusión, dado que nuestras estimaciones para **rsize** son correctas, y nos hemos apegado al funcionamiento de las regiones de memoria a través de la ejecución a partir del método raíz podemos concluir que el valor de **memRq** calculado será una sobre-aproximación de la memoria requerida.

5.1.1. Limitaciones del modelo

A pesar de ser totalmente correcto para nuestras estimaciones, el modelo desarrollado tiene algunas limitaciones. En esta sección mostraremos como en algunos casos la estimación que realiza de la cantidad de memoria necesaria para ejecutar un método es sobre-aproximada, y en esos casos, aunque nuestras maximizaciones sean exactas estaremos difiriendo del resultado real debido al planteo del modelo en sí mismo.

Las diferencias se manifiestan, por ejemplo, cuando se consideran las cadenas consecutivas, es decir, el valor de la captura hasta un determinado punto de una cadena más larga y luego la cadena completa (del estilo $m_i - m_j$ y $m_i - m_j - m_k$) consideramos la suma ya que ambas regiones estarán activas al mismo tiempo para algunos valores de los parámetros, hasta que se realiza la captura de cada una. Por lo tanto, podría pasar -y en efecto ocurre- que la máximización de una de las cadenas se dé para un determinado valor de un parámetro intermedio de las llamadas (como ser una variable inductiva de algún ciclo previo al llamado de los métodos) que haga que la otra de las cadenas no alcanze el máximo teórico planteado simultáneamente.

Veámoslo con un ejemplo. Consideremos el siguiente pequeño fragmento de código:

```
m1(int N) {
    for(int h = 1; h <= N; h++) {
        m2(h);
    }
}

m2(int k) {
    B b = new B[11 - k];
    for(j = 1; j <= k; j++) {
        m3(k);
    }
}

m3(int c) {
    for(int i = 1; i <= 5*c; i++) {
        A a = new A();
    }
}
```

Asumamos que el parámetro N tiene como precondición que es menor o igual que 10.

Como hemos visto, según el modelo para obtener la estimación de memoria deberíamos obtener los **rsize** de $m1$, $m1\_m2$ y $m1\_m2\_m3$ y como no hay bifurcaciones en el árbol simplemente sumar todos los valores.

Veamos que ocurre con la cadena m1_m2. La captura de este método consiste en el arreglo de elementos B llamado b que tiene un tamaño de $11 - k$ elementos. El valor

de la variable k se mueve en cada llamado de $m1$ con la variable inductiva h en el ciclo principal de dicho método. Por lo tanto, $11 - k$ tendrá su máximo valor cuando $k = h$ sea menor, en este caso 1. En esa oportunidad se reservarán (y luego capturarán) 10 elementos.

Por otro lado, veamos que pasa con la cadena m1_m2_m3. Si lo analizamos de manera independiente al caso anterior, observaremos que la mayor captura se da cuando c es más grande, es decir, cuando j vale k en $m2$. Por lo tanto, el valor que maximiza k será cuando h vale su máximo valor que en este caso es N. Finalmente, podemos observar que lo capturado será $5 \cdot N$.

Pero para que se dé este máximo hay que notar que el valor de h deberá ser el máximo valor, N y no 1 como en el caso anterior (salvo en el caso puntual en que $N = 1$). Podemos concluir entonces, que no habrá una iteración del ciclo de $m1$ donde simultáneamente se den ambos máximos lo cual resulta en una sobreaproximación de la memoria estimada producida por sumar sin tener en cuenta estas condiciones.

En el gráfico 5.2 se ilustra esta situación. Las primeras dos barras corresponden a las estimaciones de **rsize** de cada una de las cádenas para ciertos valores de h (diferentes). La tercera barra corresponde a la suma de las estimaciones, que será el valor que arrojará el modelo. Finalmente, podemos ver en la última barra que en una iteración dada (donde $h = k$) se dará el máximo real y efectivo, que será menor que el estimado.

5.2. Resolución del modelo de memoria requerida

Como hemos detallado, el problema a resolver posee una gran cantidad de restricciones (resolución simbólica, buena complejidad temporal en la práctica, etc), lo cual resultó en que al enfrentarnos a su resolución la cantidad de alternativas que encontramos para evaluar fue baja.

En particular, cabe destacar que la mayoría de los métodos clásicos para atacar problemas de optimización no-lineal son numéricos, contrariamente a lo que es necesario en este caso (resolución simbólica). Adicionalmente, la necesidad de obtener resultados en tiempo de ejecución decarta muchos métodos que no poseen dos etapas claramente diferenciadas que podrían ser ejecutadas previa y posteriormente a la ejecución (con la segunda parte extremadamente liviana con respecto a su complejidad de cálculo).

Más allá de la técnica elegida para la resolución del problema, consideramos solamente una técnica alternativa que cumplía en gran parte con los requerimientos. Esta técnica es la

Inicialmente, evaluamos la posibilidad de maximizar directamente los polinomios de Ehrhart dado que estos polinomios tienen algunas características especiales que podían hacer posible una maximización simbólica no tan compleja. En principio, no hay muchos resultados que traten estos temas, pero si grandes sospechas acerca de algunas propiedades.

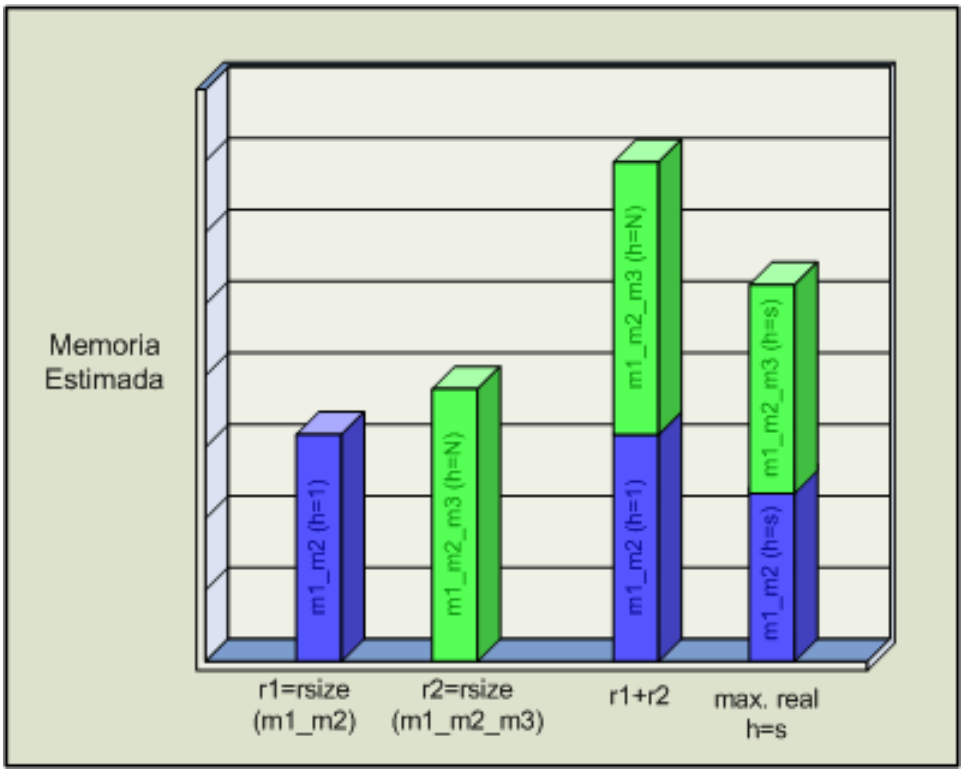


Figura 5.2: Estimaciones de memoria y cálculo real

Por un lado, los polinomios de Ehrhart se definen sobre dominios adyacentes de los parámetros. En cada uno de estos dominios, el número de puntos enteros son monótonos con respecto a uno de los parámetros (creciente, decreciente, o constante). Este hecho no ha sido probado aún, pero existen ciertas sospechas de que es cierto y de que puede ser probado.

Entonces si es posible concluir que los polinomios de Ehrhart son monótonos sobre un dominio de parámetros, entonces el máximo es necesariamente alcanzado en el borde entre dos dominios de los parámetros. Un detalle adicional es que las cotas de los dominios son racionales, y nosotros buscamos valores enteros. Parecería que esto es solucionable simplemente utilizando partes enteras, pero el desarrollo no ha sido hecho.

Finalmente, las razones por las cuales decidimos descartar este enfoque son las siguientes:

- Como hemos comentado, el enfoque aún no ha sido teóricamente demostrado y no ha sido explorado en profundidad.
- Por otra parte, los dominios que se utilizan en el problema de memoria aparecen a partir de poliedros definidos por los invariantes, por lo cual los dominios a analizar

son altamente complejos y todos estos conceptos no son directamente aplicables a ellos.

5.2.1. Aplicación de las bases de Bernstein

Introduzcamos la aplicación de la técnica con un pequeño ejemplo:

Consideremos el siguiente fragmento de código:

```
void m0(int n)
{
    for(int z = 1; z <= n; z++) {
         m(z);
    }
}

void m(int k)
{
    for(int i = 1; i <= k; i++) {
        for(int j = 1; j <= i; j++) {
            A a = new A();
        }
    }
}
```

Supongamos que deseamos, en principio, conocer la cantidad de veces que el flujo del programa pasará por la instrucción *new* dado un valor del parámetro k (lo cual es obviamente $\frac{(k+1)\cdot k}{2}$).

Dentro del cuerpo del ciclo, el invariante que se cumple es:

$$I_m = 1 \leq i \leq k \wedge 1 \leq j \leq i$$

Por lo tanto, utilizando la técnica de conteo obtenemos que los vértices del poliedro son:

$$\binom{1}{1}, \binom{k}{1} \text{ y } \binom{k}{k}$$

y la cantidad de puntos es:

$$cp_m(k) = \frac{k^2 + k}{2} = \frac{k(k+1)}{2}$$

Ahora, si quisiéramos calcular, por ejemplo, la máxima cantidad de memoria capturada por el método m en alguna de sus ejecuciones, necesitaríamos ver cual de los llamados

desde $m0$ produce más iteraciones. Por lo tanto debemos calcular el valor de n, para el cual la función $cp(k)$ nos da el mayor valor.

Los invariantes en esta cadena de llamados serían:

$$I_{m_0} = 1 \leq z \leq n$$

Uniéndolo con el de m debemos enlazar las variables z y k y unir los invariantes.

De esta manera, nuestro poliedro de entrada para Bernstein es:

$$P_{m0\_m}(\{z,k\},\{n\}) = I_{m_0} \text{ U } \{z = k\}$$

Y el polinomio será $cp_m(k)$ el cual indica la cantidad de reservas de memoria. Aplicando la transformación de Bernstein para $m0$ con invariante P_{m0_m} y polinomio $cp_m(k)$ se obtienen los siguientes coeficientes:

$$c_0 = 1$$

$$c_1 = \frac{1+3n}{4}$$

$$c_2 = \frac{n^2+n}{2}$$

Dado que se trata de restricciones con un solo parámetro podemos fácilmente conocer quién es el máximo y quién es el mínimo.

Por lo tanto, dado un valor del parámetro n, se reservarán $\frac{n^2+n}{2}$ elementos de tipo A en la ejecución del código de ejemplo. Como mínimo, se reservará un solo elemento.

5.2.2. Calculando `memRq`

Nuestro objetivo es generar una expresión fácil de evaluar (en tiempo de ejecución) en términos de los parámetros del método que puede ser usada antes de ejecutar el método para saber sus requerimientos de memoria.

Específicamente nos gustaría generar automáticamente una función para cada método que tenga la capacidad de evaluar su consumo. Idealmente, esta expresión sería un polinomio pero, como vamos a mostrar después, se vuelve un tanto más complicado.

El problema surge porque en virtud de obtener tal expresión tenemos que resolver dos problemas de maximización de manera simbólica.

En primer lugar, asumamos que podemos obtener un polinomio cada vez que aplicamos la función **rsize**. Luego, en virtud de reducir el número de variables involucradas, asumimos que las expresiones `size(Type)` son reemplazadas por el tamaño de cada tipo en la arquitectura correspondiente. Por simplicidad nosotros elegimos `size`(T)=1 para todo T.

Con el objetivo de calcular **memRq** de un m´etodo, necesitamos el grafo de llamadas del programa a partir de ese método. Luego tenemos que elegir el máximo polinomio entre los **memRq** de cada m´etodo que este llama.

Es fácil decidir que polinomio es más grande cuando lo evaluamos, pero es muy difícil sin conocer el valor de los parámetros.

Aún teniendo un procedimiento simbólico de decisión, podría pasar que un polinomio es más grande que otro en cierto dominio y el otro es más grande para otro. Por ejemplo, consideremos $P_1(n) = n^2$ y $P_2(n) = 3n + 1$. Luego $\forall n \in I\!N \cdot P_1 > P_2 \iff n > 3$.

Por lo tanto, con el objetivo de mantener la precisión debemos dejar algunas expresiones con máximos no resueltos para ser evaluadas en tiempo de ejecución o al menos generar una función que evalúa diferentes polinomios dependiendo de los argumentos de la función. Por ejemplo, una aplicación con un grafo de llamados como el de la Figura 5.3 arroja la siguiente expresión:

$$\begin{aligned}\texttt{memRq}_{m0}^{m0} = \texttt{rSize}_{m0}^{m0} + \\ max(\texttt{rSize}_{m0}^{m0,1.m1} + max(\texttt{rSize}_{m0}^{m0,1.m1,1.m3}, \texttt{rSize}_{m0}^{m0,1.m1,1.m4}), \\ \texttt{rSize}_{m0}^{m0,2.m2} + max(\texttt{rSize}_{m0}^{m0,1.m2,1.m5}, \texttt{rSize}_{m0}^{m0,1.m2,1.m6}))\end{aligned}$$

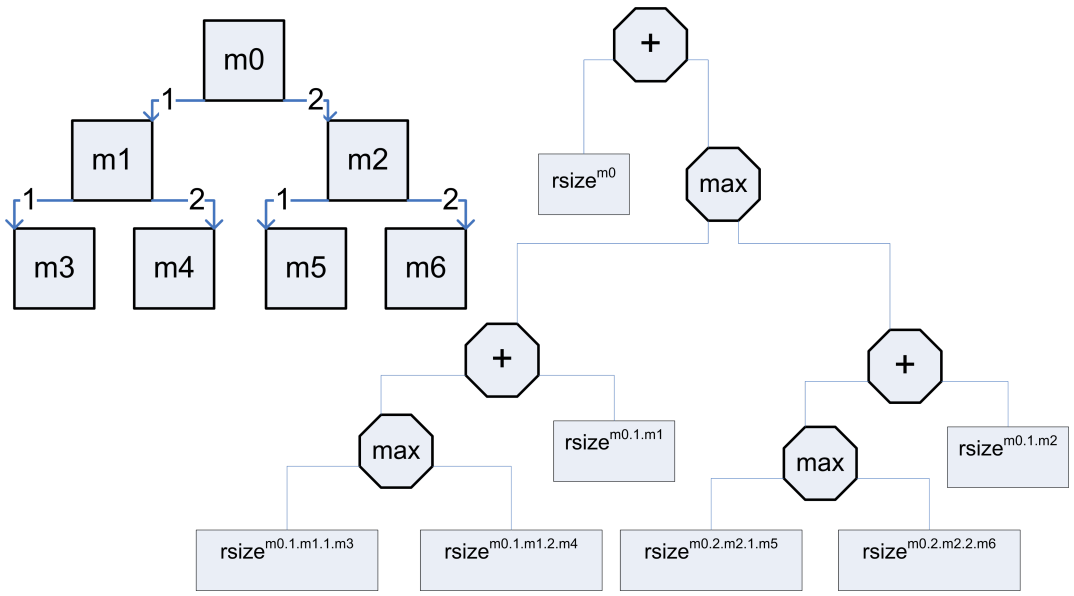


Figura 5.3: Un grafo de llamados y su circuito de evaluación subyacente

Usando esta idea, podemos automáticamente generar el código que puede ser evaluado en tiempo de ejecución y predecir los requerimientos de memoria futuros de el método elegido.

Aunque trae algo de overhead a la ejecución, la complejidad del peor caso de la evaluación es conocida a priori y además en la práctica, el tamaño del árbol de evaluación es mucho mas pequeño, dado que la mayoría de las comparaciones de máximo pueden ser resueltas de manera simbólica. La complejidad será descripta en el apéndice correspondiente (A.1.2).

5.2.3. Calculando `rSize`

```
  void m0(int m) {          void  m1(int k) {
1:  for(c=0;c<m;c++) {      1:  for (i=1;i<=k;i++) {
2:    m1(c);                2:    A a = new A();
3:    B[] m2Arr=m2(2*m-c); 3:    B[] dummyArr= m2(i);
   }                              }
}                           }
B[] m2(int n) {
1:  B[] arrB = new B[n];
2:  for (j=1;j<=n;j++)
3:    B b = new B();
4:  return arrB; }
```

$\texttt{memCaptured}(m0) =$	$\frac{3}{2}m^2 + \frac{m}{2}$
$\texttt{memCaptured}(m1) =$	$\frac{1}{2}k^2 + \frac{3}{2}k$
$\texttt{memCaptured}(m2) =$	n

$\mathcal{I}^{m_0}_{m0,2.m1,3.m2} =$	$\{0 \leq c < m, k = c, 1 \leq i \leq k, n = i\}$
$\mathcal{I}^{m_0}_{m0,2.m1} =$	$\{0 \leq c < m, k = c\}$
$\mathcal{I}^{m_0}_{m0,3.m2} =$	$\{0 \leq c < m, n = 2m - c\}$

Figura 5.4: Ejemplo y parámetros de entrada

Recordemos que **rsize** debería sobre aproximar el tamaño máximo de la región asociada con una invocación de un método y su contexto de llamada. Esto significa que necesitamos resolver un problema de optimización no lineal con el polinomio `memCaptured` representando la entrada y el invariante para la pila de control como la restricción. [1]

Como hemos señalado, la resolución simbólica off-line del problema permitiría evitar cálculos caros en tiempo de ejecución. Veamos ahora como encaramos la resolución.

La entrada del algoritmo es un polinomio P (`memCaptured` en este caso), un dominio D (el invariante de la pila de llamados) y un conjunto de parámetros $p_1, \ldots, p_n$ (los parámetros del método raíz) y este arroja una lista de dominios $D_1, \ldots, D_k$ y para cada D_i muchos polinomios (en términos de $p_1, \ldots, p_n$) representando candidatos para ser cotas superiores e inferiores de P para D_i.

Por ejemplo, las expresiones obtenidas para el ejemplo de la Figura 5.4 son:

[1] Recordar que representamos la pila (contexto de llamada) con una pila de control (labels) más un invariante que representa los posibles valores de las variables en la pila.

$\texttt{rSize}_{m0}^{m0} =$	$\texttt{max\{Bernstein}(\mathcal{I}_{m0}^{m0}, \texttt{memCaptured}(m0))\}$
Dominio:	$\{true\}$
Candidatos:	$\{\frac{3}{2}m^2 + \frac{m}{2}\}$
$\texttt{rSize}_{m0}^{m0,2.m1,3.m2} =$	$\texttt{max\{Bernstein}(\mathcal{I}_{m0,2.m1,3.m2}^{m0}, \texttt{memCaptured}(m2))\}$
Dominio:	$\{m \geq 2\}$
Candidatos:	$\{m-1, 1\}$
$\texttt{rSize}_{m0}^{m0,2.m1} =$	$\texttt{max\{Bernstein}(\mathcal{I}_{m0,2.m1}^{m0}, \texttt{memCaptured}(m1))\}$
Dominio:	$\{m \geq 1\}$
Candidatos:	$\{\frac{3}{4}m - \frac{3}{4}, \frac{1}{2}m^2 + \frac{1}{2}m - 1, 0\}$
$\texttt{rSize}_{m0}^{m0,3.m2} =$	$\texttt{max\{Bernstein}(\mathcal{I}_{m0,3.m2}^{m0}, \texttt{memCaptured}(m2))\}$
Dominio:	$\{m \geq 1\}$
Candidatos:	$\{m+1, 2m\}$

Esta solución, aunque es una buena aproximación para resolver nuestro problema de optimización, tiene algunas desventajas.

En primer lugar, el resultado no es simplemente un polinomio representando el valor máximo. Podría arrojar diferentes polinomios para diferentes dominios. Esto significa que para evaluar esta expresión es necesario decidir primero cuál es el dominio que vale para los valores de entrada. Esto significa que el orden de la evaluación está relacionado con el número de dominios obtenidos por el método. Este número está acotado por el número de vértices del dominio original.

Observar que la salida de Brenstein debe ser evaluada o procesada para obtener un valor (o expresión). Este problema es similar al máximo para **memRq** y puede ser tratado de manera análoga. Esto es, agregando los polinomios al circuito de evaluación.

Para este ejemplo, podríamos automáticamente decidir qué polinomio entre los candidatos es el más grande y finalmente computar **memRq**.

5.2.4. Reduciendo el costo de evaluación

Con el objetivo de reducir el costo de evaluación, es necesario achicar el circuito de evaluación. Esto puede ser logrado aplicando técnicas simbólicas o asumiendo algo de pérdida de precisión en las cotas superiores.

En algunos casos, es posible determinar simbólicamente el máximo entre los polinomios (arrojando directamente el más grande, o una función partida de polinomios).

En general, algunos casos que podemos manejar son:

- Por un lado, el caso lineal ya que es de simple resolución dado que es posible plantear el sistema de inecuaciones correspondientes a cada una de las hipótesis de máximo y verificar cual de ellas vale.
- Por otro lado, hemos realizado algunas inferencias para el caso de único parámetro, observando los grados de cada uno de los polinomios, analizando su paridad y su signo para, al menos, dar una cota que sirva para algunos valores (al menos, los valores "grandes").

En el peor caso, un nuevo polinomio, más grande que todos los involucrados, puede ser derivado, por ejemplo, tomando el coeficiente más grande de cada grado de los polinomios. Por ejemplo, dado $P_1(n) = n^2$ y $P_2(n) = 3n + 1$ podemos elegir $P3(n) = n2 + 3n + 1$ cuya evaluación sobre aproximará $P1$ y $P2$.

A continuación veremos la resolución de los casos lineal y de único parámetro.

5.2.5. Caso lineal

El caso lineal es de resolución medianamente fácil, ya que es posible plantear el sistema de inecuaciones correspondiente y hacer un análisis de los dominios.

Dados n coeficientes lineales $c_1, c_2, \ldots, c_n$, planteamos inecuaciones proponiendo cada uno de los c_i como máximo. Es decir, para $i = 2$ planteamos:

$$c_2 \geq c_j$$
$$\forall\ 1 \leq j \leq n, j \neq 2$$

Planteadas estas inecuaciones, utilizamos las funciones de *Polylib* para realizar el siguiente proceso:

1. En primer lugar, se generan las inecuaciones correspondientes al máximo propuesto para cada uno de los coeficientes.
2. Luego, esta información se combina con el dominio que estamos considerando en la iteración actual y se verifica si existen puntos que satisfacen todas las restricciones viendo si la intersección de los dominios es o no vacía.

Veamos un (muy simple) ejemplo:

Tomemos como entrada el polinomio:

$$P(n, m) = n$$

Sobre el politopo:

$$1 \leq n \leq m,\ p \geq m,\ q \leq 1$$

De esta manera obtenemos dos dominios, en primer lugar:

$$Q \geq P$$
$$P \geq 1$$

y los coeficientes P y 1.

Si proponemos como máximo (de manera estricta) el número 1, esto implica generar una inecuación:

$$1 > P$$

Realizando la intersección con el dominio original, que indicaba que $P \geq 1$ obtenemos un dominio vacío lo cual implica que el coeficiente 1 no es máximo en ninguna parte del dominio.

En cambio, si proponemos como máximo a P, lo cual implica la restricción $P > 1$, obtenemos en la intersección que en casi todo el dominio (salvo cuando P vale 1) éste será el máximo. En particular, cuando P vale 1, ambos son máximos (e iguales).

Para el caso del mínimo el razonamiento es idéntico salvo que se plantean las inecuaciones por menor.

El resultado para el otro dominio es similar.

Debemos notar un importante detalle. Utilizando esta técnica podría pasar que varios dominios den distintos que vacío, por el hecho que ya hemos explicado de que podría pasar que varios polinomios sean máximos en distintos intervalos. En la implementación actual, cada uno de estos coeficientes será propuesto como posible máximo.

Realizando un análisis más profundo, no sería demasiado complicado obtener estos dominios que surgen de la intersección, dando como resultado los polinomios máximos y mencionando los dominios que se obtienen. De esta manera, se podría dar al usuario la posibilidad de elegir el máximo valor dependiendo de los valores de los parámetros.

5.2.6. Único parámetro

El caso de un único parámetro es manejable dentro de ciertos parámetros. Básicamente, lo que hace la herramienta en este caso es tratar de estimar cuál será el máximo coeficiente comparando los coeficientes de mayor grado (de los coeficientes de la salida de la herramienta).

En el caso de que uno de los polinomios sea de mayor grado que el resto, ese será nuestro máximo propuesto. En otro caso, se compararán los coeficientes de los términos de mayor grado. Siempre es necesario analizar el signo y la paridad de los coeficientes, ya que si tiene un signo negativo obviamente ese polinomio probablemente no sea el máximo. Por otra parte, analizamos el dominio del parámetro porque en el caso de que sea negativo o positivo (y el máximo grado imapr) se invierte la comparación.

Claramente, este procedimiento que hemos mencionado es una heurística, que funciona mejor cuanto más grande sea el valor a evaluar. En valores pequeños puede dar resultados incorrectos (ya que podrían pesar más los coeficientes de menor grado), pero

lo planteamos como una sugerencia hacia el usuario quién deberá decidir si sus datos de entrada para los parámetros son utilizables para ésta estimación.

Veamos un pequeño ejemplo:

Dado el polinomio $P(n,m) = n^3+n$ sobre el dominio $1 \leq n \leq m$ y $p \geq m$, obtenemos los coeficientes:

$$\frac{2}{3} + \frac{4}{3}p$$
$$\frac{1}{3} + p^2 + \frac{2}{3}p$$
$$2$$
$$p^3 + p$$
$$\frac{1}{3} + p^2 + \frac{2}{3}p$$
$$\frac{2}{3} + \frac{4}{3}p$$
$$2$$
$$2$$
$$\frac{2}{3} + \frac{4}{3}p2$$

sobre el dominio $p \geq 1$.

Al ser p positivo el máximo (propuesto por nuestra heurística) es el coeficiente que posee el mayor grado, es decir $p^3 + p$. Como hemos mencionado para algunos valores podría pasar que algunos de los coeficientes de grado 2 superen a éste.

Capítulo 6

Implementación

Dada la ausencia de implementaciones de la transformación de Bernstein sobre poliedros en múltiples variables, encaramos el desarrollo de la primera implementación.

El desarrollo de la técnica ha pasado por varias etapas. La etapa inicial consistió en obtener las herramientas necesarias para solucionar ciertas partes del algoritmo, como ser la búsqueda de vértices del politopo paramétrico y el manejo simbólico de las expresiones algebraicas.

Habiendo definido estas herramientas, comenzamos a hacer las primeras pruebas sobre ejemplos fijos. Tomamos algunos pequeños códigos como entrada, calculamos sus invariantes (politopo) y consumo de memoria (polinomios) y realizamos con las herramientas los procesos necesarios para obtener los coeficientes. Hicimos algunas pruebas a mano de evaluación de parámetros que resultaron satisfactorias.

Luego, realizamos la implementación genérica del algoritmo en sí, y las primeras implementaciones de las lecturas de entrada y generación de la salida tomando como salida el conjunto de coeficientes obtenidos. Podemos ver un esquema de la arquitectura de esta implementación en la figura 6.1, puesta en contexto con las herramientas que proveen sus entradas (que aunque no han sido descriptas en profundidad sirven para ubicar la implementación en el proceso general).

Teniendo esta base implementada, pasamos a explorar la búsqueda del mínimo y máximo coeficiente, desafío que aún continua abierto para varios casos. Implementamos soluciones para el caso lineal y el caso de un único parámetro.

Como consecuencia de los intentos de resolución del problema de búsqueda del máximo, se planteó la posibilidad de aplicación recursiva del método, lo que trajo la mejora de la capacidad de procesar polinomios (en la entrada) con parámetros en sus coeficientes (ya que esta técnica realiza una aproximación al uso de Bernstein que requiere explícitamente de soporte de parámetros dentro del polinomio de entrada). Esto trajo un importante desarrollo y una extensión en el formato de entrada.

Hasta aquí podemos considerar una versión funcional y completa de la implementa-

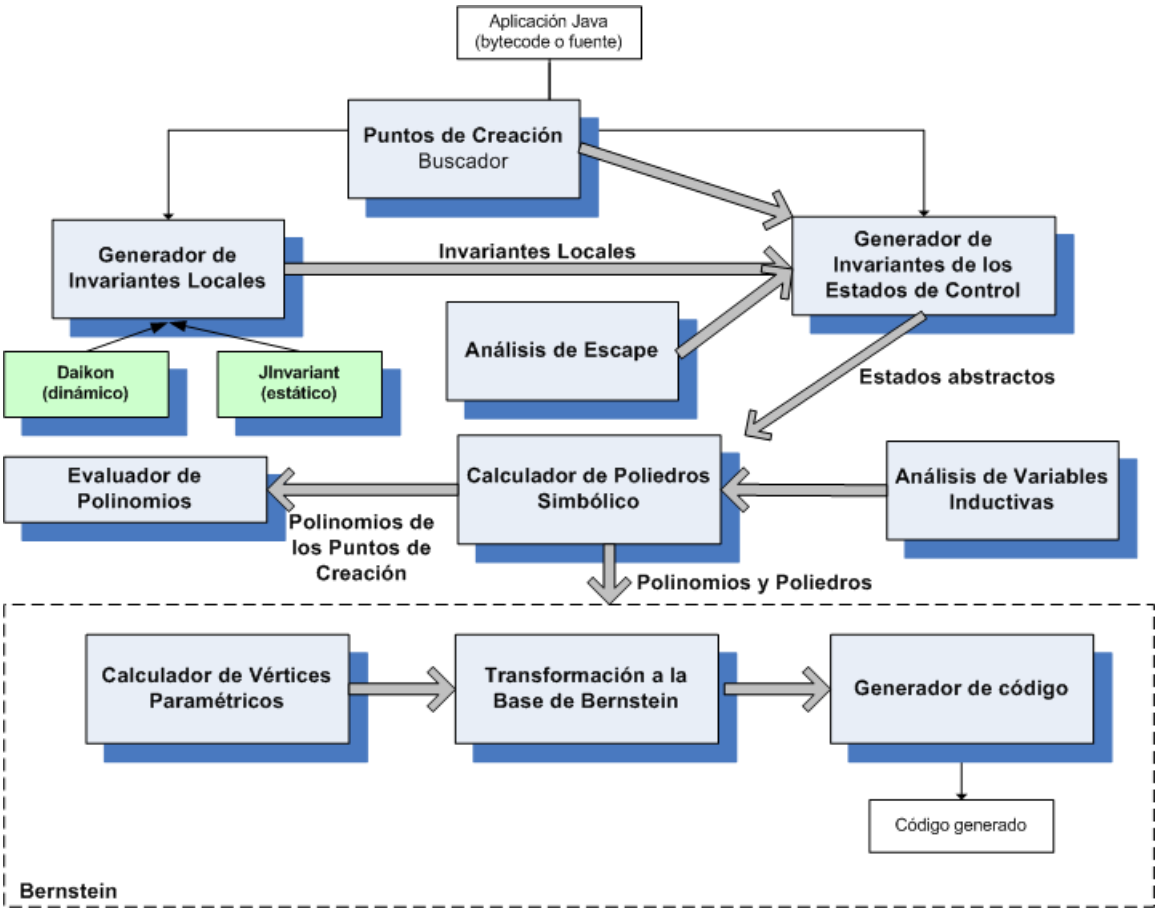


Figura 6.1: Arquitectura de la implementación de Bernstein en contexto

ción de bases de Bernstein sobre poliedros. Teniendo esta versión se pudieron realizar numerosas pruebas y todo el trabajo con las aplicaciones a memoria. En esta etapa también se realizó una integración inicial al paquete utilizado para el cálculo de memoria ([VSB+04]) con el fin de utilizar la salida de este programa como entrada de Bernstein (ya que es el flujo natural de los datos dado por el proceso).

Posteriormente, comenzamos a recibir la colaboración de Sven Verdoolage con quien comenzamos una nueva etapa de reingeniería y mejora de la aplicación. Los cambios más importantes fueron solución de algunos bugs, creación de una librería para ser llamada externamente desde C/C++ (libbernstein), integración con la herramienta de conteo Barvinok (y su distribución) y finalmente integración con el soft de cálculo Octave.

6.1. Sobre los formatos de entrada

Dada la idea primigenia de realizar una fuerte integración de la herramienta con *Polylib* [Loe99] y el uso de esta librería para la búsqueda de los vértices paramétricos, decidimos plantear un formato de entrada similar al utilizado por ésta, al menos en el caso del politopo (*Polylib* no soporta trabajar con polinomios -que no sean pseudo-polinomios de Ehrhart-).

Por lo tanto, la entrada para el politopo paramétrico debe ser similar a las funciones que computan los vértices paramétricos o la que computa los polinomios de Ehrhart. Esta entrada consiste en 2 matrices, la primera define las restricciones de las variables y la segunda define las restricciones sobre los parámetros. Debemos notar que este último tipo de restricciones pueden ser también ingresadas en la primera matriz (y no en la segunda). Cada matriz debe ser precedida por dos valores enteros representando el número de filas y columnas de la matriz. La primera columna de cada matriz contiene o bien un 0 o bien un 1. Un 0 significa que la restricción en dicha fila es una igualdad. Un 1 significa que dicha restricción es una desigualdad de la forma $linear\_equation \geq 0$. La siguientes columnas corresponden a las variables, luego los parámetros y finalmente los valores constantes.

Con respecto a la definición del polinomio, al no estar afectados por ninguna dependencia podría haberse definido de cualquier manera, pero en un primer momento para facilitar la integración con *Polylib* se decidió la siguiente forma. Una matriz puede definir todos los monomios de la siguiente manera. Cada fila está asociada a un monomio. Las primeras columnas representan a las variables, y el valor representa el grado en que aparece en dicho monomio. Las últimas dos columnas tienen que ver con el coeficiente del monomio. Dado que este coeficiente debe ser racional, la primera columna es utilizada para definir el número y la última el denominador. Los valores en esta última columna deben ser más grandes o iguales que 1. Finalmente, la matriz tiene que estar precedida por dos valores que indican el número de columnas y filas de la matriz. Notar que el número de columnas debe ser el número de variables más 2, y tiene que ser coherente con el número de variables en la definición del politopo paramétrico.

Cabe destacar que actualmente no se encuentra implementado el soporte para pseudo-polinomios pero está considerado dentro de las líneas de trabajo futuro que serán expuestas luego.

Por ejemplo, el archivo de entrada del ejemplo visto en el capítulo anterior es:

1. Definición del politopo:

```
4 5

#  i  j  N  cst
1  1  0  0  0
1 -1  0  1  0
1  0  1  0  0
```

```
1  1 -1  0  0

0 3
```

2. Definición del polinomio:

```
3 4

# i  j  num  den
  2  0  1    2
  1  0  1    2
  0  1  1    1
```

A continuación describiremos algunas modificaciones y mejoras que se han ido realizando sobre la entrada de la herramienta.

6.1.1. Integración inicial con Barvinok

La técnica de síntesis de memoria requiere del conteo de los puntos enteros de los poliedros representados por los invariantes. Por razones que exceden a este trabajo, se utiliza la solución de *libbarvinok* para el conteo (por sobre la versión de *Polylib*).

Dado que esta librería ofrece su salida solamente como una cadena de caracteres, encaramos la modificación de la etapa de salida para que convierta el o los polinomios resultantes en un formato entendible por la herramienta. De esta manera, realizamos algunas funciones para convertir polinomios en formato *evalue* (una estructura de tipo árbol para representar expresiones algebraicas), en el formato de *GiNaC* (una librería de manejo de álgebra simbólica que más adelante describiremos) para luego ser impresas de esta manera.

6.1.2. Coeficientes con parámetros

Con el objetivo de tener la posibilidad de aplicación recursiva de la técnica vimos la necesidad de soportar la aparición de parámetros en los coeficientes del polinomio.

Desde el punto de vista del cálculo, no se deben realizar grandes cambios ya que en el momento de hacerse las transformaciones los parámetros simplemente se ignoran.

Con respecto al formato de entrada, la representación del polinomio se multiplica, ya que es necesario definir un nuevo polinomio en función de los parámetros para cada coeficiente del polinomio original.

Creemos que se entiende más claramente a través de un ejemplo. La siguiente es la representación del polinomio $i^2 \cdot (N^2 + 5N) + iN + 5j$. Las variables son i, j y los parámetros N, M.

```
5 6
#  i  j  N  M cst
1  1  0  0  0  0
1 -1  0  1  0  0
1  0  1  0  0  0
1  1 -1  0  0  0
1  0  0  0  1  0

0 4

N M

3 2
# i j
 2 0
 1 0
 0 1

2 4
# N M num den
 2 0  1   1
 1 0  1   5

1 4
# N M num den
 1 0  1   1

1 4
# N M num den
 0 0  5   1
```

6.2. Sobre la salida

La salida de la herramienta pretende ser por el momento informativa para el usuario y no tiene características especiales.

En todos los casos se presentan los coeficientes de bernstein correspondientes, y en caso de obtener algún máximo o mínimo por alguno de los métodos también es impreso.

Por otra parte, la herramienta utilizada para el manejo del álgebra simbólica nos permite la producción directa de código fuente LaTeX que puede ser procesado e incorporado directamente en documentos en formato matemático.

6.3. Algoritmo

Una programa que calcule los coeficientes de Bernstein de un polinomio definidos sobre un politopo convexo debe considerar dos tipos de información como entrada:

- la definición de politopo convexo paramétrico sobre el cual las variables están definidas;
- la definición del polinomio de múltiples variables.

A través de esta descripción del algoritmo iremos ilustrando los distintos pasos con un ejemplo simple para observar cuales son los resultados intermedios. El ejemplo es:

$$P = \{1 \leq n \leq m \wedge m \leq p\}$$

$$Pol(n) = n^3 + n$$

donde las variables son n y m y p es el parámetro.

Una entrada en el formato estándar para la herramienta podría ser:

```
3 5

#s  n  m  p  cte
 1  1  0  0   -1  # n >= 1
 1 -1  1  0    0  # m >= n
 1  0 -1  1    0  # p >= m

0 3

p

2 3
#n  num  den
 3   1    1    n^3
 1   1    1    n
```

Dada esta entrada, iremos contando algunos detalles más sobre el proceso de la técnica que contamos en el capítulo anterior.

Para esta entrada, el programa debe primero llamar a la función que encuentra los vértices paramétricos del politopo (`Polyhedron2Param_SimplifiedDomain()`). La función

devuelve dominios de parámetros adyacentes, cada uno asociado a coordenadas paramétricas de los vértices. Estos vértices se guardan en una estructura para estar listos para ser utilizados en la expansión.

En este caso obtenemos:

```
Vertices:
  [ p,  p ]
  [ 1,  p ]
  [ 1,  1 ]
```

El próximo paso consiste en considerar cada dominio de parámetros. En este caso obtenemos un solo dominio el cual es $p \geq 0$.

Para cada dominio, los vértices correspondientes son usados para transformar el polinomio inicial en uno nuevo cuyas variables son los coeficientes en la combinación convexa de los vértices. Para ello primero se realiza el producto matricial entre las variables y los parámetros para luego reemplazar los vértices en el polinomio.

El producto matricial da:

```
[[a1+a2+a0*p,a1*p+a2+a0*p]]
```

El polinomio luego es transformado reemplazando la variable correspondiente por la combinación obtenida en el paso anterior (`replaceVariablesInPolynomial()`). En este caso v_0 representa la primera variable que es n.

```
Replacing: v0 by a1+a2+a0*p
```

Luego, realizamos el reemplazo y expandimos el polinomio resolviendo algunas operaciones:

```
Preliminar Expansion:
  a1+a0^3*p^3+3*a1*a2^2+3*a0*p*a2^2+3*a1^2*a0*p+
  a1^3+a2+6*a1*a0*p*a2+3*a1*a0^2*p^2+a2^3+a0*p+
  3*a0^2*p^2*a2+3*a1^2*a2
```

El próximo paso consiste en crear la base (`getBasis()`), es decir, tomar las variables y elevarlas. Se expande resolviendo algunas cuentas:

```
Basis: a1+a0+a2
Powered Basis: (a1+a0+a2)^3
Expanded Basis:
  3*a0*a2^2+3*a1*a0^2+3*a1*a2^2+a0^3+3*a0^2*a2+
  3*a1^2*a0+a1^3+a2^3+6*a1*a0*a2+3*a1^2*a2
```

Luego el polinomio es transformado nuevamente (`powerMonomials()`) con el objetivo de ser una suma de monomios de máximo grado (n). Debemos tener en cuenta también que n es el máximo de los sumas de los grados de cada fila que define un monomio en el polinomio inicial. Así obtenemos la expansión final:

```
Final Expansion:
  2*a0*a2^2+a0^3*p^3+2*a0^2*p*a2+a1*a0^2+a0^3*p+
  6*a1*a2^2+4*a0*p*a2^2+4*a1^2*a0*p+a0^2*a2+2*a1*a0^2*p+
  2*a1^2*a0+2*a1^3+8*a1*a0*p*a2+3*a1*a0^2*p^2+2*a2^3+
  4*a1*a0*a2+3*a0^2*p^2*a2+6*a1^2*a2
```

El siguiente paso consiste en interpretar los resultados obtenidos a la luz de la expansión, es decir, comparar los coeficientes de cada uno de los coeficientes de la base que habíamos obtenido para ver cuales son los coeficientes finales. En este caso obtenemos:

```
Coefficients:
        a0*a2^2         --> 2/3+4/3*p
        a1*a0^2         --> 1/3+p^2+2/3*p
        a1*a2^2         --> 2
        a0^3            --> p^3+p
        a0^2*a2         --> 1/3+p^2+2/3*p
        a1^2*a0         --> 2/3+4/3*p
        a1^3            --> 2
        a2^3            --> 2
        a1*a0*a2        --> 2/3+4/3*p
        a1^2*a2         --> 2
```

Ahora debemos analizar los coeficientes obtenidos en estas última base (`getCoefficients()`), en virtud de obtener cotas inferiores y/o superiores. Si solamente existe un parámetro, el mínimo y máximo es realmente fácil de determinar (para muchos casos). Si los coeficientes son lineales también es posible obtenerlos. En otros casos, hemos delineado varias alternativas para atacar el problema.

Veamos un pseudocódigo:

```
poliedro p;
```

```
polinomio pol(n);

dominios, vertices = obtenerVertices(p, pol);
para cada d en dominios:
    combinaciones = obtenerCombinacionVertices(d, vertices);
    reemplazarCombinacionPolinomio(pol, combinaciones);
    elevarMonomiosALaN(pol, n);
    expandirVariables(pol);
    coeficientes = intepretarCoeficientes(pol);
    minimo, maximo = buscarMaximoMinimo(pol, coeficientes);
    si se encontro maximo o minimo:
        imprimir maximo o minimo;
    sino:
        imprimir coeficientes;
```

6.4. Sobre la elección de herramientas

Dada la imposibilidad (compartida con todo proyecto de desarrollo) de implementar completamente todo lo necesario, buscamos herramientas para resolver dos partes importantes de la implementación:

1. La búsqueda de los vértices paramétricos del poliedro.
2. El manejo simbólico de las expresiones algebraicas.

El primer punto fue resuelto con la librería *Polylib*. Esta herramienta posee funciones para el manejo paramétrico de poliedros, en gran parte utilizados en las rutinas de conteo de puntos enteros (que arrojan los polinomios de Ehrhart). Con estas funciones, nos fue posible, dada una matriz que representa el poliedro de entrada, obtener sus vértices paramétricos en cada uno de los dominios correspondientes. Adicionalmente, como la intención de la implementación siempre fue lograr una integración con *Polylib* todo coincidió perfectamente.

La resolución del segundo punto fue un tanto más complicada. Necesitábamos un paquete de manejo de álgebra simbólica que en lo posible fuera fácil de integrar en un futuro dentro de *Polylib*, que sea Open Source y que sea accesible como biblioteca desde C o C++.

Finalmente, dimos con *GiNaC* [Tea06a] una herramienta de cálculo simbólico desarrollada en C++ que permite un manejo totalmente natural de las expresiones algebraicas dado que sobrecarga en la sintaxis del lenguaje las expresiones. Veamos un pequeño ejemplo:

```
int main()
{
    symbol x("x"), y("y");
    ex poly;

    for (int i=0; i<3; ++i)
        poly += factorial(i+16)*pow(x,i)*pow(y,2-i);

    cout << poly << endl;
    return 0;
}
```

Este ejemplo crea dos símbolos (variables) x e y, y les asigna un nombre. Luego, va agregando a un polinomio distintos términos. Aquí se ve como de una manera totalmente natural se pueden construir expresiones.

Al ejecutar este pequeño programa obtenemos el siguiente resultado:

```
$ c++ hello.cc -o hello -lcln -lginac
$ ./hello
355687428096000*x*y+20922789888000*y^2+6402373705728000*x^2
```

Dado que cumplía casi todos los requerimientos (salvo que no es posible llamarla desde C, lo cual complica un poco la integración con *Polylib*) decidimos utilizar la herramienta.

6.5. Sobre conversiones de datos

En cualquier aplicación que tenga grandes procesamientos de cálculos y de números es fundamental el tratamiento de los números de manera tal de no perder precisión e introducir errores.

Polylib posee varias alternativas para el manejo de grandes números con distintos tipos de datos. De todas ellas, la mejor es el uso de la librería GMP [Tea06b] que posee tipos de datos que soportan longitud arbitraria y un manejo eficiente.

Por otro lado, *GiNaC* está basada en *CLN* [Hai06] que es una librería numérica para *C++* basada en *GMP*.

Dadas estas diferencias, fue necesario durante la implementación realizar varias conversiones de tipos de datos para manejar los distintos tipos de matrices y expresiones generados a partir de estas clases base numéricas.

Lamentablemente, aún no hemos realizado las conversiones suficientes y en algunas casos fue necesario pasar por tipos de longitud no arbitraria (con la consecuente posible pérdida de precisión). Mejoras en este área constituyen una línea de trabajo futuro.

6.6. Obtención del máximo para el problema de memoria

Como hemos visto, la salida del cálculo de las bases de Bernstein puede arrojar en muchos casos un conjunto de coeficientes los cuales no entran dentro de las categorías en las que son de fácil comparación (por ejemplo, todas son lineales).

Claramente, se plantea el problema de obtener el máximo coeficiente de Bernstein (para hallar finalmente la cota). Dado que no hemos obtenido, hasta el momento, un método general para resolver este problema, debemos tomar otro enfoque.

Recordemos nuestro objetivo. Necesitamos obtener una cota superior para el uso de memoria de un método dado, *antes* de ejecutarlo. Por otro lado, es requerimiento indispensable para ser utilizado en entornos de tiempo real o embebidos que al momento de ejecución la evaluación sea sumamente rápida y sobretodo en un tiempo acotado calculable de antemano.

Con el fin de obtener el máximo, es posible atrasar su resolución generando código Java (o de algún tipo de lenguaje de programación) dejando pendiente la instanciación de sus parámetros. De esta manera, dejaríamos planteada una nueva función (el programa) que en el momento adecuado (cuando existan los parámetros) pueden ser finalmente evaluada.

Es importante destacar que esto es posible debido a que el número de evaluaciones está acotado y además podemos equilibrar la precisión en relación a la velocidad de ejecución aproximando algunos de los máximos, como hemos mencionado en 5.2.4.

Uniendo todas estas ideas decidimos separar el proceso en dos partes. Por un lado, se realiza el cálculo de los coeficientes de Bernstein, obteniéndose el listado de coeficientes. Al mismo tiempo, se genera el código necesario para ser evaluado en tiempo de ejecución, calculando el máximo cuando los parámetros ya estén instanciados. Finalmente, al momento de la ejecución, es posible agregar llamados al código original hacia el código generado (que sería compilado previamente) y ser usado para obtener las cotas.

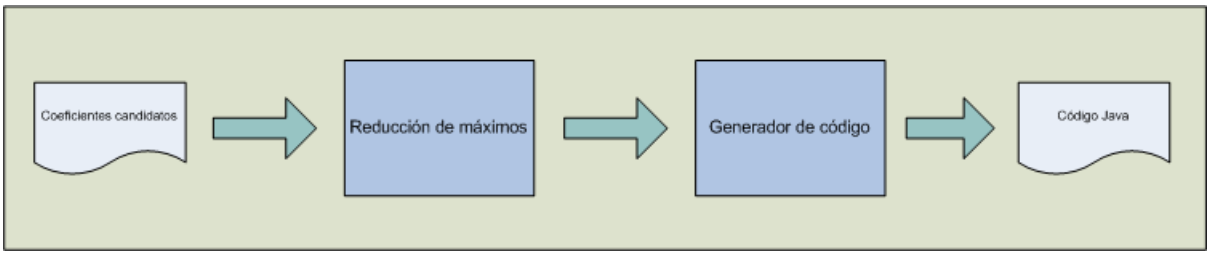


Figura 6.2: Esquema de módulos de la generación de código

En la figura 6.2 podemos ver un esquema de ésta solución.

De esta manera, se utilizan todos los datos existentes y se aprovecha al máximo los tiempos de computación.

6.6.1. De coeficientes a código Java

Dado un conjunto de coeficientes c_i que dependen de ciertos parámetros p_j queremos obtener el código Java necesario para obtener el máximo valor M del conjunto de coeficientes para ciertos valores de los parámetros de entrada.

Básicamente, es necesario evaluar cada uno de los coeficientes c_i con los valores dados de los parámetros de entrada y comparar para así obtener el valor máximo.

Teniendo en memoria los coeficientes calculados por `bernstein` es necesario transformarlos a un formato "operable". Con esto queremos decir, una cadena de caracteres que represente las operaciones necesarias para calcular la evaluación de cada uno de los coeficientes para una determinada valuación de las variables.

Para ello es necesario tener un "parser" que tome valores en el formato utilizado por `bernstein`, el cual son expresiones de la librería GiNaC y que devuelva valores en formato de cadena de caracteres que representen instrucciones de operación en lenguaje Java y puedan ser parte del código fuente.

Obtenido esto, podemos fácilmente generar el código necesario para construir un vector de resultados de evaluaciones, al cual se le asignan los distintos cálculos de las especializaciones de cada coeficiente c_i y finalmente se comparan con un ciclo de búsqueda de máximo.

Veamos un ejemplo. Dados los coeficientes $c_1 = p_1^2 + p_2 * p_1$, $c_2 = p_1^3$ y $c_3 = 0$ se obtiene el siguiente código:

```
double[] values = new double[3];
int maxIndex;

values[0] = (p_1 * p_1) + (p_2 * p_1);
values[1] = p_1*p_1*p_1;
values[2] = 0.0;

maxIndex = 0;
for(int i = 0; i < 3; i++) {
    if(values[maxIndex] < values[i]) {
        maxIndex = i;
    }
}
return values[maxIndex];
```

Notemos que:

- La "traducción" hacia código Java de las operaciones es lo más simple posible con el fin de ser multilenguaje. Por ejemplo, en el caso de las potencias, directamente se expande en las multiplicaciones sucesivas, sin llamar a métodos o funciones específicas de un lenguaje. Más allá de que el método actual está orientado a programas Java, sería muy fácil adaptarlo a otros lenguajes similares (como C++).

- Varias optimizaciones son posibles de realizarse sobre este código generado para mejorar la velocidad de la evaluación del máximo. El objetivo principal en ésta etapa es la claridad del código, pero en un futuro se podrían integrar optimizaciones (como *unroll* de ciclos) en la generación de código.

6.7. Integración con distribución de *libbarvinok*

Como hemos mencionado en la sección 6.1 el método completo requiere contar puntos enteros y la herramienta utilizada para dicha tarea es la librería *Barvinok*. Más allá de la integración inicial que realizamos, al tomar luego contacto con su autor, Sven Verdoolage, realizamos una integración completa de la técnica de bernstein en la librería.

Ahora es posible llamar al método `evalue_bernstein_coefficients` desde la distribución general y realizar el cálculo completo de los coeficientes de Bernstein desde el conteo de puntos hasta las bases.

6.8. Integración con *Omega*

Como resultado de la colaboración con Sven Verdoolage surgió esta nueva aplicación de la herramienta.

Para facilitar el uso de la misma, se la integró con el paquete de cálculo Omega [HPSSL06] y se puede llamar a las funciones de Bernstein a través de un comando del programa. Veamos un ejemplo:

```
$ ./occ
# Omega Calculator v1.2 (based on Omega Library 1.2, August, 2000):
symbolic n;
# symbolic n;
#
bmax { [i,j] -> 1/2 *i*i + 1/2*i +j : 0 <= i <= n and 0 <= j <= i };
maximize j+1/2*i^2+1/2*i over { Sym=[n] [i,j] : 0, j <= i <= n && 0 <= j }
coefficients: (1*n >= 0 && 1 >= 0) ?
(max(max(max(max(max(0.0,(3.0/2.0)*n+(n*n)/2.0),(1.0/4.0)*n),n/2.0+(n*n)/2.0),
(3.0/4.0)*n),n+(n*n)/2.0)) : 0
maximum: (1*n >= 0 && 1 >= 0) ? ((3.0/2.0)*n+(n*n)/2.0) : 0
# bmax { [i,j] -> 1/2 *i*i + 1/2*i +j : 0 <= i <= n and 0 <= j <= i };
```

Es decir, que con la función `bmax` es posible calcular los coeficientes de Bernstein llamando a las rutinas integradas ahora en *libbernstein*.

Capítulo 7

Experimentación y resultados obtenidos

Habiendo transitado el camino descripto en los anteriores capítulos, en los cuales planteamos formalmente nuestro problema, propusimos un modelo de solución y aplicación de la técnica elegida para lograr nuestro objetivo, nos ocuparemos ahora de realizar las pruebas correspondientes para empíricamente verificar si lo planteado coincide con la realidad dada por las ejecuciones.

Teniendo ya la herramienta implementada con la posibilidad de realizar los cálculos de *rsize* (mediante las bases de Bernstein) y el conjunto de aplicaciones necesarias para generar código y evaluar (necesarias para calcular *memReq*) nos disponemos a tomar algunos ejemplos, obtener todos los datos de entrada necesarios, aplicarle la técnica y finalmente instrumentar el código original y mediante ejecuciones reales comparar los resultados obtenidos en el contexto del modelo planteado.

En esta experimentación nos planteamos intentar abarcar un espectro que aunque no llegue a ser ni amplio ni mucho menos completo, si sea representativo e interesante por las características del método que explora.

En primer lugar, deseamos validar los resultados de las estimaciones de memoria dadas por la herramienta. Para ello realizaremos las ejecuciones sobre la herramienta y, como ya hemos mencionado, instrumentaremos el código para obtener , en ejecuciones reales, los números comparativos. Luego, trataremos de observar en experimentos reales la cantidad de evaluaciones que deben hacerse en *tiempo de ejecución* e intentaremos en algunos casos bajar esa cantidad resolviendo máximos a través las técnicas mencionadas en el capítulo anterior.

7.1. Ejemplo inicial

Recordemos primero el ejemplo visto en el capítulo de resolución en la Figura 7.1.

```
  void m0(int m) {          void  m1(int k) {
1:  for(c=0;c<m;c++) {     1:  for (i=1;i<=k;i++) {
2:    m1(c);               2:    A a = new A();
3:    B[] m2Arr=m2(2*m-c); 3:    B[] dummyArr= m2(i);
    }                            }
  }                            }
  B[] m2(int n) {
1:  B[] arrB = new B[n];
2:  for (j=1;j<=n;j++)
3:    B b = new B();
4:  return arrB;
  }
```

Figura 7.1: Ejemplo Inicial

Veamos, ahora, cuales son los pasos necesarios para que la herramienta pueda ser aplicada sobre este ejemplo:

1. Analizar los puntos de solicitud de memoria del código y cuales de ellos son capturados al finalizar la ejecución de cada uno de los métodos.

2. Generar los invariantes necesarios y realizar el conteo de puntos enteros con el fin de obtener la expresión de *memCaptured* para cada uno de los métodos, obviamente de forma paramétrica.

3. Obtener las cadenas de ejecución dentro del árbol de llamados, asociándolo con los invariantes y el polinomio obtenido en el conteo.

Como resultado de este proceso (que no estrictamente parte de la implementación con la que aquí estamos experimentando) obtenemos los polinomios y las cadenas de invariantes:

$$\begin{array}{|ll|}\hline \texttt{memCaptured}(m0) = & \frac{3}{2}m^2 + \frac{m}{2} \\ \texttt{memCaptured}(m1) = & \frac{1}{2}k^2 + \frac{3}{2}k \\ \texttt{memCaptured}(m2) = & n \\ \hline \end{array}$$

$$\begin{array}{|ll|}\hline \mathcal{I}^{m0}_{m0,2.m1,3.m2} = & \{0 \leq c < m, k = c, 1 \leq i \leq k, n = i\} \\ \mathcal{I}^{m0}_{m0,2.m1} = & \{0 \leq c < m, k = c\} \\ \mathcal{I}^{m0}_{m0,3.m2} = & \{0 \leq c < m, n = 2m - c\} \\ \hline \end{array}$$

Dados estos datos, podemos generar los archivos de entrada necesarios para poner en marcha la herramienta. Por un lado, debemos obtener *rsize* para cada una de las cadenas planteadas, y por otro, debemos utilizar este cálculo para generar el código Java que resuelve los máximos, teniendo en cuenta el grafo de llamadas del programa.

Veamos algunos archivos de entrada para este ejemplo. La entrada completa esta compuesta de varios grupos de archivos:

- El grafo de llamados.
- Las poliedros y polinomios necesarios para la aplicación de Bernstein a fin de obtener los rsize de los métodos.
- Los valores de los parámetros de entrada y la descripción de las cadenas existentes, junto con las definiciones del nodo raíz del árbol que no debe ser procesado a través de Bernstein dado que no necesita ser optimizado.

Para este ejemplo el archivo de grafo de llamados es el siguiente:

```
m0
m0_m1() -> m0
m0_m1_m2() -> m0_m1()
m0_m2() -> m0
```

Veamos un ejemplo del archivo para el calculo de *rsize* para una de las cadenas (m0_m1):

```
#chain_EjemploSimple_m0_00019c00009_EjemploSimple_m1_
6 8
 0  1 -1  0  0  0  0  0
 0  0  0  1  0 -1  0  0
 0  0  0  0  1 -1  0  0
 1  0  1  0  0  0  0  0
 1  0 -1  0  0  1  0 -1
 1  0  0  0  0  0  0  1

1 4
 1  0  0  1

m_init Type_B_arr

3 3
# k_init n d
 1 1 1 # k_init
 1 1 2 # 1/2*k_init
 2 1 2 # 1/2*k_init^2
```

Finalmente, el archivo que describe los parámetros, el polinomio raíz y las cadenas:

```
m0
m_init * m_init * (3.0/2.0) + (1.0/2.0) * m_init;
4
Type_A 1
m_init 10
k_init 0
Type_B_arr 1
3
m0_m1
m0_m1_m2
m0_m2
```

Salida

Dados estos datos de entrada, la herramienta genera la salida correspondiente a través de los programas descriptos en el capítulo anterior. Esto incluye llamar a Bernstein para cada cadena y generar el código correspondiente, generar el código principal Java que calcula el máximo a través de los recorridos en el árbol de llamados y la instanciación de los parámetros de entrada.

Veamos la salida para este ejemplo:

```
import java.util.*;
import java.io.*;

public class EvaluateBound {
    static int Type_A = 1;
    static int Type_B_arr = 1;
    static int k_init = 0;
    static int m_init = 10;

    public static void main(String[] args) {
        double m0 = m_init * m_init * (3.0/2.0) + (1.0/2.0) * m_init;
        double max = m0 + max(m0_m1() + m0_m1_m2(),m0_m2());
        System.out.println("Max: " + max);
    }

    public static double max(double a, double b) { return a > b ? a : b; }

    public static double m0_m1() {
        double[] values = new double[3];
        int maxIndex;

        values[0] = (3.0/4.0)*m_init-(3.0/4.0);
        values[1] = (m_init*m_init)/2.0+m_init/2.0-1.0;
        values[2] = 0.0;

        maxIndex = 0;
```

```
        for(int i = 0; i < 3; i++) {
            if(values[maxIndex] < values[i]) {
                maxIndex = i;
            }
        }
        return values[maxIndex];
    }

    public static double m0_m1_m2() {
        double[] values = new double[3];
        int maxIndex;

        values[0] = m_init-1.0;
        values[1] = 1.0;
        values[2] = m_init-1.0;

        maxIndex = 0;
        for(int i = 0; i < 3; i++) {
            if(values[maxIndex] < values[i]) {
                maxIndex = i;
            }
        }
        return values[maxIndex];
    }

    public static double m0_m2() {
        double[] values = new double[2];
        int maxIndex;

        values[0] = m_init+1.0;
        values[1] = 2.0*m_init;

        maxIndex = 0;
        for(int i = 0; i < 2; i++) {
            if(values[maxIndex] < values[i]) {
                maxIndex = i;
            }
        }
        return values[maxIndex];
    }
}
```

Esta salida es compilada con *javac* y se ejecuta para obtener el resultado final.

7.1.1. Ejemplos numéricos

La ejecución de la salida anterior da como resultado 218 (para valor de entrada 10). Veamos como podemos llegar de manera analítica a este resultado para su verificación.

En primer lugar calculemos la expresión que caracteriza la memoria capturada por el método inicial, es decir, $m0$.

El método $m0$ no realiza reservas de memoria, pero llama a otros dos métodos, $m1$ y $m2$. En el caso de $m1$ al ser un método sin valor de retorno no afecta en nada a la ejecución de $m0$ en cuanto a la memoria capturada; en cambio, $m2$ sí devuelve un arreglo de objetos del tipo B que es asignado en cada iteración del ciclo. Estos arreglos, por lo tanto, son recolectados al final de la ejecución de $m0$.

Veamos cual es el tamaño de estos arreglos en función del parámetro de entrada n, que para este ejemplo tomaremos como 10. Observando $m2$ vemos que el arreglo devuelto tiene un tamaño de n, que es el parámetro de $m2$. En conclusión, si $m0$ llama a $m2$ con el parámetro $2m - c$ la cantidad de memoria que capturará estará dada por:

$$\sum_{i=0}^{9} 2 * 10 - i = 10 \cdot 2 \cdot 10 - \sum_{i=0}^{9} i = 200 - 45 = 155$$

Como ya hemos comentado cuando hablamos de la función *memReq*, es necesario calcular los *rsize* y los distintos máximos dentro del árbol. Es decir, *rsize* de:

- $m0$
- $m0 \rightarrow m1$
- $m0 \rightarrow m1 \rightarrow m2$
- $m0 \rightarrow m2$

Y luego obtener el máximo siguiendo el grafo de llamados, calculando:

$$memReq(m0) = rsize(m0) + max(rsize(m0\_m1 + m0\_m1\_m2, m0\_m2)$$

Calculemos entonces *rsize* de m0_m2. Aquí la captura esta dada por la instrucción m2.3, ya que la memoria reservada en el new de la linea 1 escapa al método. Por lo tanto, como el método es llamado con el parámetro $2 \cdot m - c$ de $m0$, será llamado con los valores $20, 19, 18, \ldots$. En conclusión, la máxima región estará linealmente dada por el menor parámetro con que se llame, en este ejemplo será $c = 0$, es decir, $2 \cdot 10 - 0 = 2 \cdot 10 = 20$.

Veamos ahora el caso de m0_m1. Observando las relaciones entre los parámetros n y k observamos que la mayor captura en $m1$ se da para el mayor valor de k. Este será el mayor valor de c, que es $m - 1$, en este caso particular 9. Por lo tanto, cuando k sea 9 se capturará una unidad por cada iteración (por el *new* de A) e i unidades de memoria por el llamado a $m2$. Es decir,

$$\sum_{i=1}^{9} 1 + i = 9 + 45 = 54$$

Finalmente, analicemos que pasa con la cadena m0_m1_m2. En este caso lo capturado en $m2$ es n, que es como máximo i en $m1$, es decir 9.

Con estos datos calculados podemos obtener el máximo, es decir:

$$memReq(m0, 10) = 155 + max(54 + 9, 20) = 155 + max(63, 20) = 155 + 63 = 218$$

Lo cual es el resultado dado por la herramienta.

7.1.2. Ejecuciones reales

Para realizar verificaciones de manera más rápida, evitando todo este largo análisis, hemos modificado el código del ejemplo para que acumule durante la ejecución la memoria capturada en cada caso.

Veamos por ejemplo, en el caso de $m1$:

```
void  m1(int k) {
    acumM1=0;
    for (int i = 1; i <= k; i++) {
        A a = new A();
        acumM1+=1;
        B[] dummyArr= m2(i);
    }
    if(acumM1>maxM1)
         maxM1=acumM1;
}
```

En este caso vemos como acumulamos a la captura de $m1$ lo reservado en el new. Asimismo, al final de la ejecución se compara el acumulado contra el máximo para realizar la actualización.

Para el caso de $m2$:

```
B[] m2(int n) {
    acumM2=0;
    B[] arrB = new B[n];
    acumM1+=n;
    for (int j = 1; j <= n; j++) {
        B b = new B();
        acumM2+=1;
```

```
    }
    if(acumM2>maxM2)
        maxM2=acumM2;
    return arrB;
}
```

Aquí podemos observar como detalle interesante que se suma la variable `acumM1` con valor n dado que lo devuelto por el método $m2$ es capturado por el método anterior.

De esta manera obtenemos los resultados correctos para el caso anterior.

7.1.3. Comparación general numérica

Para confirmar la correctitud numérica en el ejemplo mencionado, hicimos un conjunto de pruebas con un grupo de parámetros. Los resultados pueden verse en la figura 7.2. Todos los resultados la de herramienta son exactamente iguales a las ejecuciones reales que hemos realizado.

	m0		m0_m1		m0_m1_m2		m0_m2		Máximo	
m	Real	**Bern.**	Real	**Bern.**	Real	**Bern.**	Real	**Bern.**	Real	**Bern.**
2	7	**7**	2	**2**	1	**1**	4	**4**	11	**11**
5	40	**40**	14	**14**	4	**4**	10	**10**	58	**58**
10	155	**155**	54	**54**	9	**9**	20	**20**	218	**218**
20	610	**610**	209	**209**	19	**19**	40	**40**	838	**838**
50	3775	**3775**	1274	**1274**	49	**49**	100	**100**	5098	**5098**
100	15050	**15050**	5049	**5049**	99	**99**	200	**200**	20198	**20198**
500	375250	**375250**	125249	**125249**	499	**499**	1000	**1000**	500998	**500998**
1000	1500500	**1500500**	500499	**500499**	999	**999**	2000	**2000**	2001998	**2001998**
3000	13501500	**13501500**	4501499	**4501499**	2999	**2999**	6000	**6000**	18005998	**18005998**

Figura 7.2: Valores reales y de la herramienta de memReq(m0)

7.1.4. Aspectos interesantes del ejemplo

Durante el desarrollo del método nos pareció particularmente interesante como la maximización se encuentra coordinada por los invariantes que corresponden a la pila de llamados; es decir, que más allá de buscar el máximo de una expresión, lo que hace el método es encontrar valores de un conjunto de variables de tal manera que las relaciones entre ellas hagan que algunas en particular obtengan más altos valores, y todo esto en forma paramétrica.

En este ejemplo podemos ver ese tipo de relaciones, por ejemplo, en el llamado que se realiza desde $m0$ a $m2$ dentro del ciclo (en la línea m0.3). La variable inductiva c de $m0$ se incrementa en cada iteración hasta el valor del parámetro de entrada. Por lo tanto la expresión $2 \cdot m - c$ se hace mas pequeña a medida que c crece. Por lo tanto, el máximo como hemos visto, lo obtendremos para el valor inicial de c. Finalmente, el método $m2$ captura (o escapa, en este caso son iguales) de manera inversamente proporcional a su

parámetro, por lo tanto, si buscamos la máxima captura esta se dará para el menor valor de la variable inductiva.

En cambio, en el caso de la llamada a $m1$, la relación es directa, y justamente a mayor valor del parámetro c se capturará más memoria en el método, lo cual llevará a la máxima región.

7.2. Ejemplos de benchmarks

Teniendo resultados satisfactorios en los casos planteados a través de ejemplos realizados específicamente para probar ciertas características de la herramienta, nos dispusimos a tomar algunos programas de conocidos benchmarks de Java para realizar algunas pruebas.

Con el objetivo de evitar el cálculo previo de descubrimiento de invariantes y conteo, utilizamos dos programas de test, *MST* y *Em3d* de los cuales teníamos calculada la entrada para Bernstein.

La simplicidad de estos ejemplos, en el sentido de que las maximizaciones solamente implican reemplazos de variables siguiendo los invariantes, hacen que como experimentación sean solamente útiles desde el punto de vista del uso y chequeo del modelo.

Veamos la aplicación del modelo a estos ejemplos, cuyos polinomios representando las capturas podemos ver en la tabla 7.1.

m	$\#\mathbf{CS_m}$	`memCaptured(m)`
mst		
MST.main(nv)	13	`size(mst.Graph)` + (`size(Integer)` + `size(mst.HashEntry)`) $*$ nv^2 + $per(nv, [1/4, 0, 0, 0])$ $*$ `size(mst.Hashtable)` $*$ nv^2 + (`size(mst.Vertex)` + `size(mst.Vertex[])`) $* nv + 5 *$ `size(StringBuffer)`
MST.parseCmdLine()	2	`size(java.lang.RuntimeException)`+`size(Integer)`
MST.computeMST(g, nv)	1	`size(mst.BlueReturn)`*(nv-1)
em3d		
Em3d.main(nN,nD)	26	`size(em3d.BiGraph)` + $nN *$ (2 $*$ `size(em3d.Node)` + 4 $*$ `size(em3d.Node[])` $*$ nD + 2 $*$ `size(double[])` $*$ nD) + 8 $*$ `size(em3d.Node1Enumerate)` + 4 $*$ `size(java.lang.StringBuffer)` + `size(java.util.Random)`
Em3d.parseCmdLine()	6	3 $*$ `size(Integer)` + 3 $*$ `size(java.lang.Error)`
BiGraph.create(nN,nD)	2	`size(em3d.Node[])` $* nN$

Cuadro 7.1: Polinomios que muestran la estimación de la captura de memoria de los ejemplos MST y Em3d.

MST

Primero reemplazemos los tamaños de los tipos de datos, ya que estamos asumiendo que todos valen 1. De esta manera obtenemos que la captura para el método *main* es $nv^2 + per(nv, [1/4, 0, 0, 0]) * nv^2 + nv + 5$, para *parseCmdLine* es constante y finalmente la función principal que es *computeMST* tiene una captura de $nv - 1$.

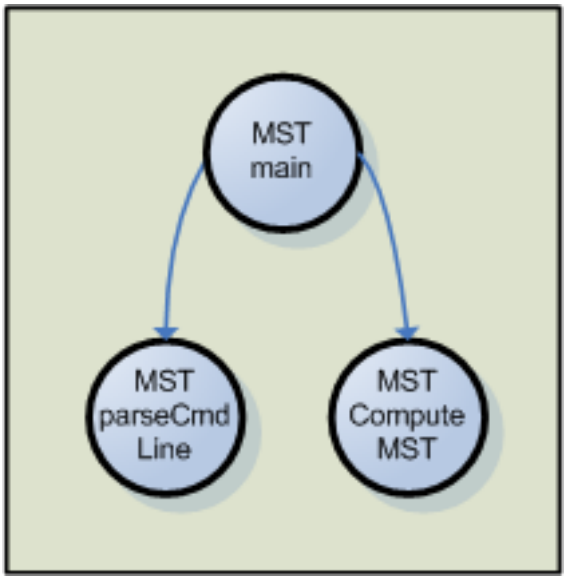


Figura 7.3: Esquema del código de MST

Veamos ahora como van a variar estos valores y como se relacionan entre sí. Teniendo conocimiento de la estructura del código (esquematizado en la figura 7.3) de estos ejemplos observamos que los parámetros nv en *main* y nv en *computeMST* están relacionados por el predicado que los iguala (el parámetro g se ignora). Por otro lado, la cadena de llamados se da de la manera en que hemos presentado los métodos en la figura. Finalmente, al existir un solo parámetro que es el correspondiente, además, al punto de entrada si queremos maximizar estas expresiones no debemos hacer más que tomar un único parámetro nV como parámetro de la maximización y dar las expresiones de cada uno de los métodos como expresión del máximo.

Esta situación se da porque no existen ciclos dentro del código en los que se hagan llamados a métodos, por lo cual los invariantes son muy simples y con simples reemplazos algebraicos podemos resolver el problema.

Em3d

Lo mismo pasa en este ejemplo. Los parámetros nN y nD se encuentran enlazados directamente, y el método *parseCmdLine* solamente aporta un término constante al cálculo, por lo tanto la expresión del máximo resultante serán dichos polinomios.

Finalmente, queremos mencionar que la aplicación de la técnica a ejemplos reales está fuertemente ligada al hallazgo de invariantes, es decir, a la preparación previa de los ejemplos para ser utilizados por la herramienta. Dados los invariantes y su captura, siempre es posible aplicar nuestro método para obtener las cotas de memoria.

7.3. Reducción de las evaluaciones

Dado el programa Java generado por el proceso (7.1), es interesante analizar las operaciones necesarias para evaluar el resultado para un determinado conjunto de parámetros.

Veamos como está conformado el código generado. Cada uno de los métodos que aparecen representa a una cadena de las del grafo de llamados, y para obtener el máximo de los candidatos deben hacerse k evaluaciones de polinomios, donde k es un número que esta directamente relacionado con la cantidad de vértices paramétricos que tiene el poliedro.

Por otro lado, luego de las evaluaciones es necesario calcular máximos entre las distintas ramas del árbol. La cantidad de estas evaluaciones claramente esta relacionada con los llamados que se realicen según el grafo de llamados. En concreto, en esta etapa se realizarán tantos máximos como bifurcaciones haya en el árbol de llamados.

En ambos casos, es posible aplicar las técnicas (exactas, como en los casos lineales, o aproximadas como en la búsqueda de los polinomios superadores) desarrolladas y explicadas para entre selección los candidatos arrojados por Bernstein.

7.3.1. Casos lineales

En el ejemplo visto, dos de las tres cadenas evaluadas tienen polinomios lineales. Por lo tanto, si en lugar de obtener todas las alternativas de Bernstein, utilizamos el módulo de resolución lineal implementado directamente obtendremos:

- **m0_m1_m2**: los polinomios obtenidos son:

$$p_1(m) = m - 1$$

$$p_2(m) = 1$$

  Y la herramienta señala que el máximo es p_1.

- **m0_m2**: los polinomios obtenidos son:

$$p_1(m) = 2m$$

$$p_2(m) = m + 1$$

  Y la herramienta señala que el máximo es p_1.

De esta manera, podemos reducir la cantidad de evaluaciones a la mitad en este ejemplo, y en mucho más en ejemplos más grandes donde los candidatos sean numerosos, con la seguridad de obtener un resultado exacto.

7.3.2. Aproximaciones y heurísticas

Sobre la otra cadena, esta compuesta de dos coeficientes, es decir, dos candidatos a ser máximos, de los cuales uno no es lineal, lo cual imposibilita la aplicación de la técnica anterior. Estos polinomios son:

$$p_1(m) = \frac{3}{4} \cdot m - \frac{3}{4}$$
$$p_2(m) = \frac{1}{2} \cdot m^2 + \frac{1}{2} \cdot m - 1$$

De cualquier manera, quedan algunas opciones para atacar el problema:

- **Heurística de único parámetro**: al tener un único parámetro es posible (aunque no de manera exacta , ni sobre aproximada en algunos casos) aplicar una heurística bajo la idea de que los monomios con mayor grado y coeficiente serán los que prevalezcan para valores medianamente grandes. Como ya hemos remarcado en varias oportunidades, esta alternativa solamente se puede utilizar cuando no es importante cometer algunos errores de supra aproximación. En este caso, al tener p_2 mayor grado sería el polinomio elegido.
- **Polinomio superador**: otra alternativa que ya hemos mencionado consiste en construir un nuevo polinomio con los mayores monomios de cada grado. Aunque probablemente nos alejemos de una cota justa, el resultado siempre es sobre aproximado. En este caso podríamos utilizar como polinomio $\frac{1}{2} \cdot m^2 + \frac{3}{4} \cdot m - \frac{3}{4}$.

7.3.3. Sobre los máximos del grafo de llamadas

A su vez, los máximos en el grafo de llamados pueden ser tratados de igual manera, como fue mencionado. Todas las técnicas descriptas pueden ser utilizadas para este nuevo problema que también aporta una mejora en la cantidad de evaluaciones que es necesario realizar, sobretodo en ejemplos grandes.

Capítulo 8

Conclusiones y trabajo futuro

8.1. Conclusiones

La principal conclusión que puede obtenerse de este trabajo es que es posible, luego de sintetizar los requerimientos de memoria de un método, obtener los picos de memoria antes de ejecutar el método, de una manera que lo hace totalmente utilizable en entornos embebidos y de tiempo real.

Los resultados de las aproximaciones que hemos obtenido son lo suficientemente cercanos al valor exacto para nuestros ejemplos. La precisión que se obtiene depende de tener buenos invariantes y de si se deja la evaluación del máximo para tiempo de ejecución.

Por otra parte, otro objetivo que ha sido logrado es el de generar una sólida implementación original (de una técnica reciente y original) que es novedosa por la cantidad de requerimientos y restricciones que satisface ya que es simbólica y para los tamaños usuales de los métodos posee una complejidad práctica razonable.

Como veremos en la sección 8.2.2 existe mucho trabajo a futuro a realizar. Sin embargo, esta tesis representa un interesante punto de partida en la resolución del problema que se plantea.

Las nuevas aplicaciones y la difusión que está tomando la librería nos demuestra que es un proyecto útil para una gran cantidad de problemas del análisis estático de programas y auguramos un futuro aún mejor en su evolución.

8.2. Otras aplicaciones

Además de la aplicación fundamental de esta tesis, es decir, el problema del conteo de memoria, nuevas aplicaciones dentro del campo del análisis de programas han surgido para la técnica y para la implementación en particular.

Cabe destacar en general, que dado que el método de Bernstein sobre cajas se puede pensar como un caso particular de la técnica generalizada a poliedros, todas las aplicaciones generales del método de Bernstein también lo son de esta implementación.

Algunas de estas aplicaciones pueden encontrarse en [CT04].

8.2.1. Aplicaciones a optimizaciones sobre Arreglos

La colaboración de Sven Verdoolaege con la implementación se dio en gran parte debido a su interés de utilizar la herramienta para algunas nuevas aplicaciones relacionadas con arreglos.

El problema que intentan resolver, el cual es bastante similar al de memoria, consiste en contar el máximo número de elementos vivos en un determinado arreglo.

En realidad, el problema original se trata de determinar el tamaño de los canales en una red de procesos.

Para una iteración dada, es posible fácilmente computar el número de elementos producidos y los elementos consumidos. La resta de lo segundo menos lo primero da el número de elementos vivos y ahora la idea es obtener la máxima cantidad sobre el dominio completo de iteraciones, pero dependiendo de los parámetros.

Claramente, la necesidad de optimizar un número de elementos, sobre un dominio de iteración (poliedro en base al invariante) plantea una aplicación para la técnica presentada en esta tesis.

Más información sobre esta aplicación puede encontrarse en [SV06].

8.2.2. Aplicaciones a predicción de ingreso a memoria caché

En su tesis de doctorado, Sven Verdoolaege trata el problema de predicción de ocupación de caché. Con la aparición de la implementación de la técnica mejora la posibilidad de saber si algún dato entrará en caché teniendo en cuenta el uso de memoria, calculando en tiempo de ejecución las funciones de análisis.

Más información de esta aplicación es posible encontrarla en [VBBC05].

8.3. Trabajo futuro

Más allá de que hemos logrado resolver el problema de la estimación de memoria que ha sido el objetivo de esta tesis, han quedado algunos aspectos de la resolución que están pendientes y por su extensión han quedado fuera del alcance.

Por un lado, existen algunos casos de entradas a la herramienta que, aunque sabemos que son fácilmente implementables, no las hemos finalizado. En este grupo se encuentran

el soporte para múltiples dominios en la maximización de la memoria y el tratamiento de polinomios de Ehrhart.

Por otro lado, sabemos que estamos dejando un problema abierto que no hemos podido aún atacar de manera certera desde el punto de vista teórico, que es el de encontrar el máximo coeficiente en la salida del cálculo de las bases de Bernstein.

Finalmente, otro aspecto del trabajo futuro es el que continuamos realizando, que se trata de la mejora de la implementación que se ha iniciado a partir de la colaboración de Sven Verdoolaege.

8.3.1. Sobre el manejo de dominios

En la explicación de la conexión entre la técnica de conteo de memoria y la aplicación de las bases de Bernstein, no hemos entrado con gran detalle en el tema de los dominios. En parte, esto es debido a que los ejemplos que hemos tomado no poseen estas complicaciones como consecuencia de que el soporte para estos casos en la implementación realizada no es aún maduro.

Intentando ser más explícitos, como salida del conteo de memoria, que básicamente se trata de un conteo de puntos enteros sobre un poliedro paramétrico, podría pasar que se divida en varios dominios dependiendo de los parámetros del poliedro. Por lo tanto, una simple cadena de entrada para Bernstein podría en realidad dividirse en distintos casos dependiendo del valor de los parámetros.

Asimismo, la salida de la implementación de Bernstein, al depender de un poliedro paramétrico, podría también dividirse en varios dominios, lo cual implicaría una generación de código más compleja y una más larga posterior evaluación.

8.3.2. Sobre el soporte de polinomios de Ehrhart

De manera análoga a lo explicado en el punto anterior, la salida de la etapa de conteo, es estrictamente hablando, un pseudo-polinomio de Ehrhart. Este podría tener coeficientes periódicos que complicarían el tratamiento por la herramienta de Bernstein.

Este problema puede tratarse de varias maneras:

- **Acotar.** Al estar realizando un conteo que busca sobre-aproximar los valores es posible simplemente tomar de cada coeficiente periódico el valor más grande. Por ejemplo, si tenemos el polinomio:

  $$[8, 10, 4]x^2 + [1, 2, 3]x + [6, 7, 8]$$

  Podemos simplemente elegir $(10, 3, 8)$ para todo valor de x. En el caso de ser paramétricos puede ser un poco más complicada la cota.

- **Aproximar.** Existe un interesante trabajo sobre la aproximación de polinomios de Ehrhart a través de polinomios normales. Refinando una conjetura dada por Ehrhart que conecta las propiedades de las caras del poliedro con la periodicidad de su polinomio de Ehrhart, se encontraron dos métodos de aproximación. Ambos son buenas aproximaciones (su error relativo tiende asintóticamente a cero) y su complejidad es polinomial si el número de variables y parámetros está fijo. Más información sobre estas ideas pueden encontrarse en [Mei04].

- **Tratar como polinomios múltiples.** Dado que los polinomios periódicos dividen el dominio en n partes (donde n es su periodicidad) es posible tratarlos como n polinomios sobre n dominios, simplemente eligiendo para cada valor de los parámetros el polinomio correcto.

  Es posible realizar esto de manera clara y simple encapsulando con una estructura conveniente este tipo de polinomios.

En los ejemplos que hemos analizado hemos evitado este tipo de polinomios, ya que existen criterios bastante simples en cuanto a los poliedros utilizados para evitar la generación de coeficientes periódicos.

8.3.3. Sobre el problema abierto del máximo y su alcance

Hemos mencionado, con cierto nivel de detalle, este problema en los capítulos previos. Básicamente, se resume a obtener el máximo polinomio de un conjunto de n (que en general es bastante chico, no más de 6 u 8 para un método normal) todos en función de los parámetros del método.

Cabe destacar que el máximo polinomio puede no ser único, ya que podría pasar que en distintos dominios de los parámetros el máximo cambie. Otro detalle del problema radica en que este máximo es necesario obtenerlo de manera simbólica ya que los valores efectivos de los parámetros se sabrán únicamente en runtime.

Lo que es importante destacar en esta sección en relación al trabajo futuro, es que:

- El alcance para esta tesis del problema del máximo es limitado, ya que hemos planteado nuestra alternativa de resolución a partir de la generación de código con gran éxito.

- El problema del máximo es un interesante problema, de gran dificultad ya que hemos consultado a varios especialistas en Ecuaciones Polinomiales y no hemos conseguido aún una respuesta que nos permita avanzar en una solución general con todas las características necesarias. Como consecuencia de esto, su resolución escapa al alcance de esta tesis pero abre una línea de investigación muy interesante que continuaremos explorando.

Finalmente, como hemos visto, hemos tenido varias estrategias para intentar resolverlo pero en cualquier caso no hemos podido solucionar el caso general. De cualquier manera, veremos ahora algunas aproximaciones que delinean lineas de trabajo futuro.

Método Sturm

Si los candidatos son polinomios en términos de un parámetro solamente, podríamos aplicar algunas técnicas como ser Sturm [Wei06] la cual dados dos polinomios arroja los dominios donde el primero es más grande que el segundo. Este enfoque tiene un problema debido a que nuestra resolución debe ser simbólica, y el método de Sturm es naturalmente numérico.

Existen técnicas clásicas como [Ped91] con un enfoque similar al segundo punto mencionado pero que soporta múltiples parámetros. Este enfoque aún no ha sido analizado en profundidad para su aplicación.

Técnicas de aplicación recursiva

Durante toda la implementación del proyecto hemos trabajado fuertemente en contacto con Philippe Clauss. Al arribar al problema del máximo la teoría desarrollada hasta ese momento no tenía respuesta al problema (es decir, en cuanto a obtener simbólicamente el máximo coeficiente). De esta manera iniciamos una etapa de exploración e investigación del tema, consultando bibliografía y especialistas en el tema. Una de las ideas más interesantes que halló Philippe Clauss es del uso recursivo de las bases de Bernstein para atacar el problema de maximización.

Daremos aquí un resumen de estas ideas (ya que aún no están completas).

Recordemos nuestro problema de maximización. Debemos hallar el máximo coeficiente, es decir, un c_i que cumpla que $\forall j, c_i \geq c_j$, o lo que es lo mismo, que $c_i - c_j \geq 0$.

Consideremos este nuevo polinomio. Podemos probar que es siempre positivo si todos sus coeficientes son positivos y sus parámetros son siempre positivos. Por otro lado, si somos capaces de calcular sus coeficientes de Bernstein deberíamos probar que son todos (o al menos el mínimo) positivos.

Para reaplicar el método debemos cumplir algunas condiciones:

- Debemos definir las nuevas variables y parámetros, y estos deben pertenecer a un politopo convexo. En este caso hay que revisar como es la relación entre cada uno de los parámetros. En este punto es donde es necesario extender la implementación (y así fue hecho) para que soporte la aparición de parámetros en el polinomio.
- Es necesario determinar como están acotadas las distintas variables entre sí. Esto puede hacerse con el algoritmo de eliminación de Fourier-Motzkin. En el mejor caso, todas las variables serán eliminadas y los coeficientes serán numéricos, pero no suele ser lo común.

Finalmente, obtenemos los coeficientes de Bernstein dependiendo de los parámetros. Si estos coeficientes son positivos, definimos condiciones para las cuales vale la desigualdad entre los coeficientes originales ($c_i \geq c_j$). Si las condiciones son lineales es posible chequearlas.

Este método fue probado para algunos ejemplos bastante simples, con resultados exitosos, pero de cualquier manera no terminan de ser una resolución general ya que siempre se depende de ir hallando la positividad de los distintos términos evaluados.

Esta es la línea de investigación futura más fuerte en la resolución de este interesante problema.

8.3.4. Sobre el futuro de *libbernstein*

El proyecto de una implementación útil, que resuelva nuestro problema, fue alcanzado, pero al llegar a este punto vimos como se fue ampliando el alcance y a su vez nuevas aplicaciones aparecían.

Esto dio como resultado que *Bernstein* se convierta en una librería de uso general con posibilidad de ser llamada desde herramientas externas, con una interfaz clara, y con integración con paquetes matemáticos (como *Octave*).

La situación actual es que la implementación de la librería se encuentra consolidada pero en activo desarrollo. Se ha integrado con el paquete de *libbarvinok* lo cual le da difusión e integra su funcionalidad y creemos que en el futuro el desarrollo avanzará hacia nuevas mejoras.

Apéndice A

Cálculo de complejidad

En este apéndice presentamos como información adicional el cálculo que hemos realizado de complejidad para las distintas partes de la herramienta. Diferenciaremos los aspectos teóricos y prácticas del cálculo, y separamos la descripción en dos etapas. En primer lugar el cálculo de las bases de Bernstein parámetricas para el cálculo de `rSize` y luego el cálculo de la estimación con el grafo de llamados (el cálculo de **memReq**).

A.1. Algoritmo de Bernstein

Como parte central del método se encuentra el algoritmo de Bernstein para poliedros convexos.

Sus entradas son:

- El conjunto de restricciones dadas por los invariantes cuyas caracteracterísticas más interesantes radican en el poliedro paramétrico resultante. En particular, su cantidad de vértices, ya que éstos estarán ligados a los pasos de la transformación. Otro punto interesante es la cantidad de variables que aparecen en los invariantes, ya que dará la dimensión de los puntos del poliedro.
- El polinomio correspondiente, donde también podemos observar las variables y el grado de cada uno de los monomios.

Todos estos parámetros serán utilizados luego como dato para algunas operaciones.

El algoritmo se divide en dos grandes partes. En primer lugar, el hallazgo de los vértices paramétricos correspondientes y luego el cálculo de los coeficientes de Bernstein en sí, utilizando estos vértices hallados.

A.1.1. Obtención de los vértices paramétricos

En esta sección mostraremos cuál es la complejidad de esta parte de la resolución del problema teniendo en cuenta nuestra implementación con *Polylib*.

Para realizar el cálculo de vértices es necesario realizar varios procesos, pero el más importante es el hallazgo de los vértices paramétricos en sí (otros son, por ejemplo, el cambio de representación del poliedro). Siguiendo los resultados de [LW97] que describen los procesos utilizados en *Polylib*, observamos que la complejidad del proceso es de:

$$O(f \cdot (m^3 + m^2 n))$$

donde f es el número de caras del poliedro, n es la dimensión del espacio de variables y m es la dimensión del espacio de parámetros. Esto se deduce del hecho de que para cada cara de grado m, se realiza una inversión de matrices ($\Theta(m^3)$)) y una multiplicación de matrices ($\Theta(m^2 n)$).

Cabe destacar que f, es decir, las caras del poliedro está en relación exponencial con respecto a las restricciones originales del poliedro. Por lo tanto, aunque la complejidad del proceso podría parecernos en un principio polinómica, cuando vemos cuál es la entrada real del proceso observamos que es exponencial.

Consideraciones prácticas

Más allá de las complejidades teóricas (no muy favorables) debemos destacar que en la práctica y para nuestro tipo de entrada el algoritmo se comporta muy bien.

Dado que los análisis que realizamos consisten en solamente algunos métodos y dado que la cantidad de parámetros y variables que aparecen en un método promedio es bajo (buenas prácticas de programación recomiendan no más de 6 - 8 variables por método, y menos parámetros aún), los invariantes que se generan son simples y con pocas restricciones, por lo que aunque se realize una exponenciación en la complejidad ésta no se alejará de valores manejables.

Pruebas empíricas del método de búsqueda de vértices paramétricos han dado buenos resultados en ejemplos prácticos. Por ejemplo, algunas pruebas en un Pentium 133 Mhz (tecnología hoy obsoleta) tomó menos de 0.03 segundos para computar los 34 vértices paramétricos de un poliedro de dimensión 4 con 4 parámetros, que posee 126 caras, y 0.22 segundos para computar los 89 vértices paramétricos de un poliedro de dimensión 5 con 5 parámetros, que posee 1170 caras.

Durante nuestras pruebas de la herramienta, con métodos reales, nunca hemos superado los 7 u 8 vértices con tiempos muy bajos. La cantidad de parámetros en general no superó los 4 o 5, y lo mismo ocurrió con las variables. Teniendo en cuenta los números anteriormente mencionados, suena totalmente lógico y razonable esperar un buen rendimiento práctico para el algoritmo.

A.1.2. Cálculo de los coeficientes de Bernstein

Habiendo visto en la sección anterior una complejidad teórica fuerte (exponencial), poco nos queda de esperar superarla en esta parte del algoritmo (que además de por sí es bastante más simple computacionalmente). De cualquier manera repasaremos las operaciones asociadas.

En general, esta sección del algoritmo realiza manipulaciones algebraicas simbólicas y algunas operaciones de múltiplicaciones entre matrices.

En concreto, recordemos cual es el algoritmo:

```
1:    polihedro p;
2:    polinomio pol(n);
3:
4:    dominios, vertices = obtenerVertices(p, pol);
5:    para cada d en dominios:
6:        combinaciones = obtenerCombinacionVertices(d, vertices);
7:        reemplazarCombinacionPolinomio(pol, combinaciones);
8:        elevarMonomiosALaN(pol, n);
9:        expandirVariables(pol);
10:       coeficientes = intepretarCoeficientes(pol);
11:       minimo, maximo = buscarMaximoMinimo(pol, coeficientes);
12:       si se encontro maximo o minimo:
13:           imprimir maximo, minimo;
14:       sino:
15:           imprimir coeficientes;
```

Analizemos el algortimo a partir de la línea número 5.

Sea m la dimensión de los vértices encontrados, v la cantidad de vértices, a la cantidad de variables del poliedro original, o la cantidad de monomios del polinomio, n es el grado del polinomio original.

En principio iteraremos de acuerdo a la cantidad de dominios obtenidas en el paso anterior. Llamemos a este valor d. Para cada uno de estos dominios realizaremos:

- `obtenerCombinacionVertices`: se realiza una múltiplicación entre los vértices obtenidos y las variables a_i que serán utilizadas para construir las combinaciones convexas. Esta operación posee una complejidad de $O(v \cdot m)$ productos.
- `reemplazarCombinacionPolinomio`: se reemplazan las apariciones de las variables por lo recién calculado. La complejidad de este cálculo es de $O(o \cdot a)$. Luego se realiza una expansión, que tiene un órden similar (o menor) al del paso siguiente.

- `elevarMonomiosALaN`: este es uno de los pasos más costosos desde el punto de vista de complejidad computacional. En este paso se pasan todos los monomios existentes a la potencia n multiplicando en cada caso. Cada una de estas expansiones tiene un costo de $O((no) \cdot (o + (o-2) \cdot 3n)))$, y deben hacerse varias operaciones de estas. En principio, podría pasar que todos los monomios sean necesarios de elevar, por lo cual deberíamos hacer tantas expansiones como monomios tengamos. Y este punto, lo peor que puede pasar es que tengamos un monomio para cada combinatoria de variables a_i (lo cual es $a_i!$). También se agregan algunas multiplicaciones para distribuir estos resultados, pero carecen de importancia en comparación de estos cálculos.

- `intepretarCoeficientes`: finalmente, debemos interpretar los coeficientes en la base que hemos expandido, lo cual implica algunos recorridos por los monomios realizando divisiones. Esto está en el orden de la cantidad de monomios que tendrá este nuevo polinomio.

En conclusión, la complejidad de esta sección del código tiene su máxima expresión cuando se realizan las distintas expansiones y se elevan a la n los monomios para obtener la expresión en la base.

A.2. Cálculo del máximo sobre el grafo de llamadas

En la segunda etapa de la ejecución de la herramienta, dados los polinomios que maximizan cada cadena es necesario realizar la evaluación de cada uno con los parámetros instanciados y luego calcular los máximos y sumas de acuerdo a lo explicado.

Dado que las sumas y los máximos se realizan entre los valores ya instanciados y evaluados podemos considerar dichas operaciones como de tiempo constante, y enfocarnos en los recorridos necesarios para encontrar las cadenas.

El grafo de llamadas se compone de *nodos* que representan los diferentes métodos del fragmento de código a analizar y *ejes* (orientados) que representan los llamados entre ellos. Si nuestro objetivo es obtener las distintas cadenas que existen en el grafo para evaluarlas (ya que así se planteó la entrada de la herramienta) lo que debemos obtener son todos los caminos (orientados) que comienzan en un cierto método m (el método a analizar).

Por lo tanto, es fundamental para el cálculo evaluar cuantos *nodos* y *ejes* tendrá.

La cantidad de *nodos* es directamente la cantidad de métodos que posee el programa. Para el cálculo que estamos haciendo éste será uno de los parámetros de entrada.

La cantidad de *ejes* es bastante más interesante, ya que depende de la estructura del programa. Lo que podemos ver es que en el peor caso teórico estaríamos frente a un completo orientado. Pero antes de avanzar debemos reflexionar acerca de que significa esto para la estructura de un programa.

Básicamente, lo que se está planteando es que cada método llama a todos los demás métodos, por lo tanto, la obtención de todos los caminos en este tipo de grafo es equivalente a considerar las combinaciones de las apariciones o no de cada uno de los ejes. Esto es efectivamente posible en un programa (y sin realizar recursión), ya que por ejemplo podríamos tener condiciones en el ingreso a cada método decidiendo cual será el próximo en ser llamado (dependiendo por ejemplo de parámetros de entrada).

Por otro lado, al iniciarse los caminos siempre en el método m deberíamos eliminar los que no comienzen en ese nodo, pero en el contexto de tantos caminos no tiene gran impacto.

En conclusión, la realización de este tipo de permutaciones llevará a una complejidad exponencial, ya que es necesario probar combinaciones (viendo si cada eje va o no va en cada una) del orden de 2^m (donde m es la cantidad de ejes).

En el aspecto teórico y de lo posible se podrían complicar aún más este modelo, ya que por ejemplo al realizar múltiples llamados con distintos o iguales valores de parámetros a los mismos métodos, los métodos llamados se duplicarían en el grafo generando así un multigrafo y aumentando aún más la complejidad.

Más allá de todos estos detalles, pasemos a ver la complejidad que tendrá el algoritmo en los casos más comunes (que fueron del tipo que nos aparecieron en aplicaciones reales) que es lo que realmente nos interesa.

A.2.1. Complejidad en la práctica

La realidad indica que los programas reales son mucho más simples que los que hemos mencionado. Es extremadamente poco común encontrar grafos de llamadas que se parezcan a un completo. En general, lo normal es que tengamos grafos de la forma que vemos en la figura A.1 (en peor caso):

Como podemos ver, todos los métodos son llamados en un cierto sentido lógico del estilo $m_i \rightarrow m_j$, donde $i < j$, y en ese sentido existen todos los ejes posibles. Es posible asumir también que hay pequeño número limitado de métodos que se llaman en algún otro orden lógico sin perder nuestra cota de complejidad.

La complejidad de obtener las cadenas en este esquema es del orden de n^2, es decir, cuadrático en la cantidad de métodos que tenga el fragmento de código.

Esto es porque la cantidad de ejes (llamados) que inciden en cada uno de los nodos sigue una serie en la que en el nodo $n+1$ llegán $k+1$ llamados, siendo k los llamados que recibía el nodo n. De esta manera, se realiza la suma de números consecutivos sobre todos los nodos que da un orden cuadrático como mencionamos.

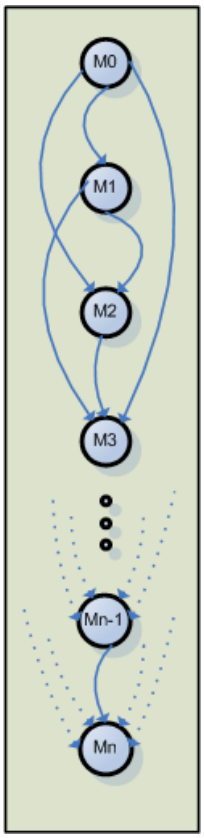


Figura A.1: Estructura del peor caso práctico

Bibliografía

[BFG+05] V. Braberman, A. Ferrari, D. Garbervetsky, P. Listingart, and S. Yovine. Jscoper: Eclipse support for research on scoping and instrumentation for real time java applications. In *ETX 2005*, San Diego, USA, October 2005.

[BGY05a] V. Braberman, D. Garbervetsky, and S. Yovine. On synthesizing parametric specifications of dynamic memory utilization. *Internal Report. Verimag, France*, 2005.

[BGY05b] Victor Braberman, Diego Garbervetsky, and Sergio Yovine. Synthesizing parametric specifications of dynamic memory utilization in object-oriented programs. In *FTfJP'2005: 7th Workshop on Formal Techniques for Java-like Programs*, Glasgow, Scotland, July 26, 2005.

[BGY06] Víctor 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, jun 2006. http://www.jot.fm/issues/issue_2006_06/article2.pdf.

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

[BW93] A. Bowyer and J. Woodwark. Introduction to computing with geometry, 1993.

[CHQR05] W. Chin, H. H.Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *SAS 05*, 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.

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

[Coo71] S. Cook. *The complexity of theorem-proving procedures*. Proc. 3rd Ann. ACM Symp. on Theory of Computing Machinery, New York, 1971.

[CT04] Ph. Clauss and I. Tchoupaeva. A symbolic approach to bernstein expansion for program analysis and optimization. In Evelyn Duesterwald, editor, *CC 2004*, volume 2985 of *LNCS*, pages 120–133. Springer, April 2004.

[Ehr77] E. Ehrhart. Polynômes arithmetiques et m´ethode des polyedres en combinatorie. *Series of Numerical Mathematics*, 35:25–49, 1977.

[Fah98] T. Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *TJS*, 12(3), 1998.

[Far93] G. Farin. *Curves and Surfaces in Computer Aided Geometric Design*. Academic Press, San Diego, 1993.

[FG96] R. T. Farouki and T.Ñ. T. Goodman. On the optimal stability of the bernstein basis. *Math. Comput.*, 65(216):1553–1566, 1996.

[FR87] R. T. Farouki and V. T. Rajan. On the numerical condition of polynomials in berstein form. *Comput. Aided Geom. Des.*, 4(3):191–216, 1987.

[FR88] R. T. Farouki and V. T. Rajan. Algorithms for polynomials in bernstein form. *Comput. Aided Geom. Des.*, 5(1):1–26, 1988.

[GCYZ04] D. Garbervetsky, C.Nakhli, S. Yovine, and H. Zorgati. Program Instrumentation and Run-Time Analysis of Scoped Memory in Java. *RV 04, ETAPS 2004, ENTCS, Barcelona, Spain*, April 2004.

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

[GJ79] M. Garey and D. Jonhson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, San Francisco, 1979.

[GW88] G.Nemhauser and L. Wosley. *Integer and Combinatorial Optimization*. John Willey & Sons, 1988.

[Hai06] Bruno Haible. *CLN*: Class library for numbers. Available at http://www.ginac.de/CLN/ , March 2006.

[HJ03] M. Hofman and S. Jost. Static prediction of heap usage for first-order functional programs. In *POPL 03*, 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*. ACM, 1999.

[HPS96] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96*, pages 410–423. ACM, 1996.

[HPSSL06] College Park High Performance Software Systems Laboratory, Univ. of Maryland. *Omega*: Frameworks and algorithms for the analysis and transformation of scientific programs. Available at http://www.cs.umd.edu/projects/omega/ , March 2006.

[Kar72] R. Karp. *Reducibility among combinatorial problems.* In R. Miller and J. Thatcher (eds.), Complexity of Computer Computations, Plenum Press, New York, 1972.

[Loe99] V. Loechner. *PolyLib*: A library for manipulating parameterized polyhedra. Available at http://icps.u-strasbg.fr/ loechner/polylib/, March 1999.

[LW97] Vincent Loechner and Doran K. Wilde. Parameterized polyhedra and their vertices. *Int. J. Parallel Program.*, 25(6):525–549, 1997.

[Mei04] Benoît Meister. *Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization.* PhD thesis, Université Louis Pasteur, December 2004.

[Ped91] Paul Pedersen. Multivariate sturm theory. In *AAECC91*, pages 318–332, London, UK, 1991. Springer-Verlag.

[Spe94] Melvin R. Spencer. *Polynomial real root finding in Bernstein form.* PhD thesis, Provo, UT, USA, 1994.

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

[SV06] H.Nikolov y T. Stefanov S. Verdoolaege. Improved derivation of process networks, 2006.

[Tea06a] GiNaC Team. *GiNaC*: A computer algebra system. Available at http://www.ginac.de/About.html , March 2006.

[Tea06b] GMP Team. *GMP*: Gnu multiprecision arithmetic library. Available at http://www.swox.com/gmp/ , March 2006.

[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.

[Val79] L. Valiant. *The complexity of computing the permanent.* Theoretical Computer Science 8, 1979.

[VBBC05] Sven Verdoolaege, Kristof Beyls, Maurice Bruynooghe, and Francky Catthoor. Experiences with enumeration of integer projections of parametric polytopes. In *CC*, pages 91–105, 2005.

[VSB+04] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In *CASES '04*. ACM, 2004.

[Wei06] Eric W. Weisstein. Sturm function., 2006.