Una herramienta para inferir Cotas Superiores de Relaciones de Coste

Diego E. Alonso Blas

Proyecto de Sistemas Informáticos

Universidad Complutense de Madrid

Facultad de Informática

Dirigido por:

Elvira Albert Albiol y **Puri Arenas Sánchez**

Septiembre de 2008

Autorización

El autor, Diego Esteban Alonso Blas, autoriza a la Universidad Complutense de Madrid a utilizar y difundir con fines académicos, no comerciales, y mencionando expresamente a su autor, tanto la presente memoria como el código, la documentación y el prototipo desarrollado.

En la Facultad de Informática de la Universidad Complutense de Madrid, Calle del Profesor García Santesmases de Madrid, en Septiembre de dos mil ocho.

Fdo. Diego Esteban Alonso Blas

Resumen

El presente proyecto desarrolla un componente de software para manejar Sistemas de Relaciones de Coste (SRC) y calcular cotas superiores en forma cerrada de sus relaciones. Los SRC son ecuaciones numéricas generadas por herramientas de análisis estático de costes. El componente los procesa para inferir de ellos cotas superiores de esos costes. También se ha desarrollado una aplicación para ilustrar esta funcionaldad. Ambos sistemas han sido desarrollados en JAVA, usando solo software libre y gratuito.

Abstract

This project developes a software component devised for handling Cost Relations Systems (CRS) and infering closed-form upper bounds for their relations. CRS are numeric equations generated by static analysis tools. Our component manipulates CRS in order to infer an upper bound of those execution costs. We have also devlopd an application in order to illustrate these features. Both systems are developed in JAVA with free (*free press* and *free beer*) software.

Palabras clave (Keywords)

- Sistemas de Relaciones de Coste *(Cost Relation Systems)*
- Análisis estático de Costes *(Static Cost Analysis)*.
- Software Libre en JAVA *(Free* JAVA *Software)*
- Sistemas de álgebra por computador. *(Computer Algebra System)*

Agradecimientos

Quisiera desde aquí agradecer a mis familiares por su cariño, su esfuerzo y su gran ayuda, gracias a la cual he podido llegar hasta aquí.

No puedo tampoco olvidar la ayuda y apoyo que me han dado las Directoras, por su iniciativa en promocionar este proyecto y por la constante atención con que me han ayudado a sacarlo adelante.

Asimismo agradezco a Samir Genaim y otros miembros del CLIP la asistencia técnica sin la cual no habria completado el proyecto.

Índice general

1. Introducción: Análisis de Coste **13**
1.1. ¿Por qué eficiencia? 13
1.1.1. Grandes posibilidades 13
1.1.2. Serias carencias 14
1.1.3. Eficiencia 14
1.1.4. Costes de ejecución 15
1.2. ¿Cómo saber? 16
1.2.1. Métodos de estudio 17
1.2.2. Fortalezas y debilidades 18
1.3. El proyecto 20
1.3.1. Problema y objetivo 20
1.3.2. Introducción a los SRC 21
1.3.3. SRC: Independencia del lenguaje 23
1.4. SRC vs SED 25
1.4.1. Indeterminismo 26
1.4.2. Imprecisión 29
1.4.3. Bucles de múltiples argumentos 29

2. Sistemas de Relaciones de Coste **31**
2.1. Introducción intuitiva 31
2.1.1. Ecuaciones y Relaciones 31
2.1.2. Ecuaciones y restricciones. 32
2.1.3. Llamadas y recursión 32
2.2. Definición formal 34
2.2.1. Preliminares 34
2.2.2. Expresiones de Coste 37
2.2.3. Relaciones de Coste 40
2.2.4. Ecuación de coste 41
2.2.5. Sistemas de Relaciones de coste 44
2.2.6. Árboles de evaluación 44
2.3. Cálculo de cotas superiores 45

2.3.1. Fórmula general 45
2.3.2. Ranking functions 46
2.3.3. Invariantes 47
2.3.4. Evaluación parcial 48

3. Especificación de Requisitos **51**
3.1. Sistemas y subsistemas 51
3.2. Requisitos Funcionales 51
3.2.1. Resolutor 51
3.2.2. Prototipo 52
3.3. Requisitos no funcionales 53
3.3.1. R.N.F. de implementación 53
3.3.2. Requisitos no funcionales del producto 55

4. Desarrollo del sistema **57**
4.1. Arquitectura del sistema 57
4.1.1. Arquitectura 57
4.1.2. Diseño Detallado del Resolutor 58
4.1.3. Diseño Detallado del parser 60
4.1.4. Diseño del prototipo 61
4.2. Herramientas usadas 62
4.2.1. JavaCC 62
4.2.2. JUnit 63
4.3. Bibliotecas reutilizadas 64
4.3.1. PPL 64
4.3.2. JGraphT 65

Presentación. Material y métodos

Este proyecto desarrolla una herramienta que procesa unos Sistemas de Relaciones de Coste para obtener cotas superiores en forma cerrada de las relaciones principales del sistema original. Busca proporcionar una nueva implementación más portable y eficiente que su predecesor, el sistema PUBS.

El capítulo 1 da una introducción a la eficiencia y los costes de ejecución, por qué se desarrollan sistemas más eficientes y cómo se pueden medir los costes de un programa. Introduce las técnicas de análisis estático y cómo éstas pueden potenciar los sistemas software y hardware, y facilitar a los ingenieros su implementación.

El capítulo 2 presenta formalmente los Sistemas de Relaciones de Coste y expone cuáles son los algoritmos esenciales para el cálculo de cotas superiores.

El capítulo 3 contiene la especificación de los requisitos software del sistema desarrollado; siguiendo la clásica separación entre funcionalidades (Requisitos Funcionales) y las directrices y reglas sobre el comportamiento del sistema y su desarrollo (Requisitos no Funcionales).

El capítulo 4 detalla la implementación del sistema, su diseño arquitectónico y detallado, las herramientas que hemos usado para construirlo y las principales bibliotecas que hemos reutiliado en este proyecto.

El disco contiene el código fuente de esta memoria y la última revisión *trunk* (número 156) del sistema desarrollado, todo ello como proyectos del IDE Eclipse. El fichero README detalla las instrucciones para su instalación y uso.

Capítulo 1

Introducción: Análisis de Coste

1.1. ¿Por qué eficiencia?

1.1.1. Grandes posibilidades

¿Quién no usa un ordenador?

En la industria actual los podemos encontrar para conducir un camión, controlar las válvulas de una refinería, coordinar los robots de una fábrica de coches y otros procesos que quisiéramos pensar.

Si observamos el trabajo en una empresa o en la Administración, veremos software con el cual inventarían sus almacenes, abonan las nóminas a sus empleados, compran sus materias primas, venden sus productos, intercomunican sus departamentos o planifican sus futuras estrategias y políticas.

Las actividades artísticas e intelectuales, como el periodismo, la música, el dibujo, el cine o la animación han renacido gracias a los aparatos y los programas que hacen más fácil crear y reproducir sus contenidos. Y éstos rápidamente se difunden por la Web.

Gracias a ello a cualquier persona un ordenador le sirve de periódico, enciclopedia, atlas, tienda, banco, teléfono, correo, álbum, radio, *jukebox*, televisor e infinidad de otros muchos usos.

Así pues a nadie escapa cómo los ordenadores y el software han cambiado y mejorado nuestro modo de vivir y de trabajar.

Y los ingenieros en informática deben proporcionar ordenadores y programas para las personas, adaptadas a éstas, sus necesidades y sus limitaciones.

1.1.2. Serias carencias

Queda poco tiempo

Como nuestra vida resulta limitada, queremos aprovechar mejor cada minuto y segundo y nos duele todo retraso inútil. Entonces, ¿cómo no nos desagradará que el ordenador tarde varios segundos en mostrar una web, varios minutos en enseñar las fotos de nuestras vacaciones, varias horas en descargar una película o varios años en plegar esa importante proteína?

No tenemos dinero

Por prósperos que seamos nunca dispondremos ilimitadamente de dinero para nuestras necesidades. Tampoco para los recursos del computador.

Un procesador más rápido, una memoria más amplia, una antena de transmisión más veloz, más energía eléctrica, la conexión a internet... demandan un gran coste, que no siempre podemos asumir.

1.1.3. Eficiencia

Para ahorrar recursos, los usuarios desean invertir solo lo necesario y conveniente para su objetivo:

- Si se emplea el computador para largos y complejos cálculos es preferible comprar un procesador más productivo.
- Quien navega o trabaja por la Web busca un ordenador óptimo para conectarse a Internet.
- Para música, películas o juegos interesa un *repository* más amplio, o una pantalla y altavoces más refinados, o un mando más intuitivo, etc.
- Si viajamos mucho optaremos por un aparato ligero y pequeño.
- Y quien tiene poco dinero quiere un ordenador asequible cuyos programas gasten pocos recursos y no engorde sus facturas.

Todos en general desean un sistema que proporcione más **más y mejores funcionalidades** y **ahorre sus recursos**.

En esencia, todos desean ordenadores y programas dotados de la máxima eficiencia.

1.1.4. Costes de ejecución

Los ingenieros deben proporcionar sistemas más eficientes, pero para ello necesitan saber valorar esa eficiencia objetivamente.

La definición anterior es muy subjetiva, pues cada persona valora y emplea de distinta manera sus recursos. Una noción alternativa y más objetiva de eficiencia la proporcionan los costes de ejecución.

Un **Coste de Ejecución** ***(Execution Cost)*** es una magnitud que cuantifica el consumo de un recurso escaso a causa de la ejecución de un programa en un ordenador. Algunos ejemplos son:

- Tiempo que tarda el programa en completar una tarea, o número de tareas que completa en un determinado tiempo.
- Instrucciones máquina ejecutadas, distinguiendo operaciones enteras y de punto flotante, saltos, accesos a memoria o llamadas al sistema.
- Bytes de memoria que ocupa, entre la región de código del programa o las regiones de datos.
- ¿Cuántos procesos e hilos se crean? ¿Qué datos comparten? ¿Cuándo necesitan sincronizarse?
- Lecturas y escrituras a memoria o a disco, así como fallos de memoria cache o de memoria virtual.
- Electricidad que consume toda la máquina o bien cada componente.
- Datos o mensajes que envía por una red de telecomunicación.
- Consumibles de E/S, como la tinta y el papel en una impresora.
- Costes asociados a requisitos: para la confidencialidad consideramos que un coste es el número de registros de datos a que se accede.
- Y otros costes ponderados, como instrucciones ejecutadas por julio de consumo eléctrico, porcentaje de lecturas a memoria que generan fallos de cache o memoria virtual, etc.

Los costes de ejecución nos permiten definir la eficiencia como la relación entre los resultados y la funcionalidad de un sistema y sus costes.

Así pues los Ingenieros deben desarrollar sistemas que implementen la funcionalidad requerida con el **mínimo coste de ejecución** posible.

1.2. ¿Cómo saber?

Interesa pues un método capaz de estudiar los costes, esto es observar qué recursos se consumen al ejecutar un programa, medir en qué cuantía y probar que ese consumo es efecto de esa ejecución,

Tal interés no es solo teórico, sino que quien desarrolla un ordenador o un programa lo precisa durante todo el ciclo de desarrollo.

- Durante la especificación estimamos si una restricción, por ejemplo "*el programa tarda menos de un segundo en responder*", es viable.
- Al diseñarlo podemos evaluar las posibles arquitecturas en función de su capacidad para cumplir esos requisitos.
- Si estamos construyendo el sistema nos guía y nos indica qué componentes actúan como motores o cuellos de botella.
- Cuando ya validamos el software, nos permite garantizar si el sistema efectivamente cumple esos requisitos.

Puesto que los costes son **comparativos** – un programa no tiene per se costes bajos sino menores que otro–, nuestro método deberá decirnos:

- ¿Cómo **comparamos** los costes de varios programas? ¿En qué unidad se **miden**? ¿Podemos **acotarlos** fielmente?
- ¿En qué **circunstancias** valen nuestros resultados? Quizás dependen del entorno físico en el que está la máquina, del tipo de máquina, de qué otros programas se ejecutan en ella...
- ¿Dependen los resultados de los datos procesados por el programa? Si hay alguna medida del tamaño de los datos, ¿cómo aumentan, o se escalan los costes para datos mayores?
- ¿Cómo verificamos los resultados de nuestros estudios? ¿Hasta qué punto podemos confiar en sus resultados? ¿Se parece nuestro método al de otra ciencia? ¿Nos permite **confirmar** o **refutar** los resultados?

1.2.1. Métodos de estudio

Existen dos enfoques principales de estudio, inspirados en otras ciencias.

Dinámico o experimental

En las ciencias naturales, como la Física o la Biología, se emplea el método hipotético-deductivo: para un fenómeno se plantea una hipótesis que lo explique y se diseña un experimento reproducible para contrastarla.

Así, para explicar cómo afecta la gravedad a objetos de distinta masa, *'supongo'* que el más pesado recibe más aceleración (hipótesis) y pruebo a dejarlos caer desde una misma altura (experimento) para comprobar si llegan a una vez o el más pesado antes (contraste y refutación).

El enfoque dinámico propone estudiar los costes con experimentos donde ejecutamos un programa en una máquina con unos datos. Por ejemplo:

- Comparamos los costes de varios ordenadores para ejecutar el mismo programa: en esto consisten LINPACK, SPEC o TCP.
- Averiguo qué programa es más eficiente midiendo cuánto gasta un mismo ordenador en ejecutar cada uno con idénticos datos.
- Descubrimos qué entradas le cuesta más tratar a un mismo programa ejecutándose sobre un mismo computador.

Análisis estático

En Matemáticas se emplea el método deductivo-axiomático, que define las propiedades de unos objetos con unos enunciados sencillos –axiomas–, de los que luego se infieren otros más complejos–teoremas–.

Por ejemplo, de los postulados de geometría de Euclides se infiere el Teorema de Pitágoras o de los axiomas de Peano se infieren las propiedades de la suma o de la multiplicación de números naturales.

El enfoque estático estudia de manera semejante el coste de ejecución de un programa: especificamos estructuralmente cualquier posible programa, suponemos un modelo axiomático del coste de cada estructura de programación y de él deducimos como teorema el coste de un programa concreto.

Así se calculan (o deducen) el total de escrituras a memoria de un programa a partir de axiomas tales como que cada asignación genera una escritura o que el coste de una secuencia es la suma de los costes de sus elementos.

1.2.2. Fortalezas y debilidades

¿Qué enfoque es más útil? Evidentemente, lo será el que mejor se adapte a las circunstancias y los desafíos de la Ingeniería Informática actual.

Diversos y curiosos relojes

Tanto ha proliferado y tanto se ha ramificado la industria hardware, que hoy *computador* no solo son *desktops*, *laptops* o servidores. Sino que también puede serlo una videoconsola, un reproductor de vídeo, un teléfono móvil o incluso una red que combine varios de estos aparatos.

Ante tal diversidad resulta costoso un estudio dinámico completo. Salvo que usemos alguna simulación, difícilmente tendremos un ejemplar de cada aparato en que ejecutar el programa.

En cambio en un análisis estático no necesitamos tener, sino solo ajustar el modelo de costes a cada máquina. Incluso podemos estudiar unos planos de una máquina, sobre los cuales no hay experimento directo posible.

Evolución del Software

Una aplicación actual se construye ensamblando decenas o centenares de componentes. Esto afecta al estudio de costes por dos razones.

- Aunque cada módulo evoluciona despacio, el programa termina acumulando muchas pequeñas evoluciones. Esto puede provocar que tener dos veces un idéntico programa sea tan difícil como lo sería bañarse dos veces en el mismo agua, pues siempre se mueve algún elemento.
- Como los costes del sistema emergen de sus componentes; interesa reconocer sus motores o cuellos de botella.

Un experimento dinámico solo estudia una versión de todo el programa, pero no informa sobre el efecto de cada componente. Así que al introducir algún mínimo cambio en el programa deberemos repetir todo el estudio.

El análisis estático es composicional: empieza por construcciones mínimas y sobre ellas deduce el coste de una función, una clase. . . y de la aplicación, de forma que el resultado refleja la estructura del programa. Así sí reconocemos los costes y el peso de cada componente. Y si uno cambia recalculamos sus costes locales y los de sus clientes y reutilizamos el resto.

Mentiras, malditas mentiras y *benchmarks*

A veces interesa acotar el coste de un programa y garantizar esa cota para cualquier dato del conjunto de entrada. Para manejar ese conjuno usualmente infinito usamos dos abstracciones:

- El tamaño, una medida numérica de la que depende el coste.
- Unas particiones lógicas de esos datos, donde el programa actúa de modo similar, aunque haya *esquinas* donde se comporte erráticamente.

Un experimento dinámico nunca garantizará cota alguna, pues necesitaría infinitas pruebas. Solo se ejecuta en algunos datos que supuestamente representan las tareas reales. Si no refinamos y actualizamos constantemente esos datos, puede ocurrir, que un sistema que inicialmente supera sus *benchmarks* luego falla catastróficamente en tareas reales[1].

Por el contrario, para un análisis estático, puesto que sus premisas no dependen de las entradas, sus resultados valen para cualquier dato de entrada. Así garantizan para algunos programas cotas de los costes en función del tamaño de los datos. Y esos resultados pueden detectar mejor las condiciones de esquina.

Just in CASE

Un ingeniero debe también ser más productivo para su empresa, lo que logra usando aplicaciones CASE[2]. Tan importantes son éstas, que ninguna técnica tiene exito sin su apoyo, y el estudio de costes no es excepción.

Para un experimento dinámico solo se necesitan: los aparatos de medida, un mecanismo de pruebas, programas para recopilar y analizar estadísticamente los resultados y los mencionados *benchmarks*. Estas herramientas son tan sencillas y asequibles que cualquier ingeniero puede diseñar y realizar un estudio dinámico. Por ello es muy utilizado actualmente.

Más difícil resulta un análisis estático. Éste precisa conocer su notación y apenas sí hay una herramienta CASE que automatice el cálculo y presente intuitivamentes resultados. Consecuentemente este enfoque es poco usado.

[1] Un ejemplo de ésto fue el cohete *Ariane 5*, que usaba el navegador de su predecesor pero volaba más rápido, lo que provocó un fallo por desbordamiento en ese chip.

[2] *Computer Aided Software Engineering.*

1.3. El proyecto

1.3.1. Problema y objetivo

El análisis estático supera mejor las dificultades actuales de hardware, software y aplicaciones. Pero no les resulta plenamente adecuado a los ingenieros de software, cuya necesidad de herramientas productivas no ha sido satisfecha hasta ahora.

Tales herramientas podrían ayudar a los programadores, proporcionando estas y otras funcionalidades:

- En la documentación de un programa, por ejemplo los JavaDocs, se podría anexar la información de los costes de ejecución de cada método.
- Depuración: el compilador señalaría si un método tiene costes muy altos, y detectaría (avisaría que no puede garantizar la ausencia de) bucles infinitos, interbloqueos...
- Elaborar gráficos y mapas de costes, que indicaran los motores y los cuellos de botella del sistema y representaran su impacto.
- Validación: Certificar una cota de costes para un programa o componente, adquirido externamente, de dudosa fiabilidad.

Por otra parte, un analizador estático podría potenciar las capacidades de los sistemas operativos:

- La planificación en tiempo real, que requiere que cada tarea termine antes de un plazo límite (*deadline*), será más eficaz si podemos predecir cuántos ciclos de procesador gastará cada tarea.
- El sistema de memoria virtual podrá evitar muchos fallos de página si se prevee cuántos accesos a memoria realiza un proceso y cómo se dispersan, y asignamos marcos de página en consecuencia.
- Conocer los costes locales de cada proceso, así como el tráfico de datos con otros procesos, es una buena orientación para repartirlos mejor entre los núcleos de un multiprocesador o los nodos de un *cluster*.

Este proyecto de grado participa en un proyecto más ambicioso cuyo objetivo es obtener y ofrecer tales herramientas, contribuyendo así a la difusión del análisis estático en el desarrollo industrial de software.

1.3.2. Introducción a los SRC

Como técnica matemática, el análisis estático precisa una representación y notación formal en que modelar los costes.

¿Qué buscamos?

La notación que escojamos será luego la base de las herramientas que creemos. Para que éstas les resulten prácticas y productivas a los ingenieros de software, nuestra notación deberá:

- Ser muy **intuitiva**, para que así quienes usan como quienes desarrollan las herramientas puedan aprenderla y dominarla.
- **Correcta**, que pueda modelar el coste de todos los posibles programas.
- El modelo debe ser **independiente** de cualquier lenguaje o paradigma de programación. Así podremos comparar dos programas sin preocuparnos de cómo hayan sido construidos.
- Definida de manera **constructiva**, para así poder representarla en un programa y manipularla con algoritmos.

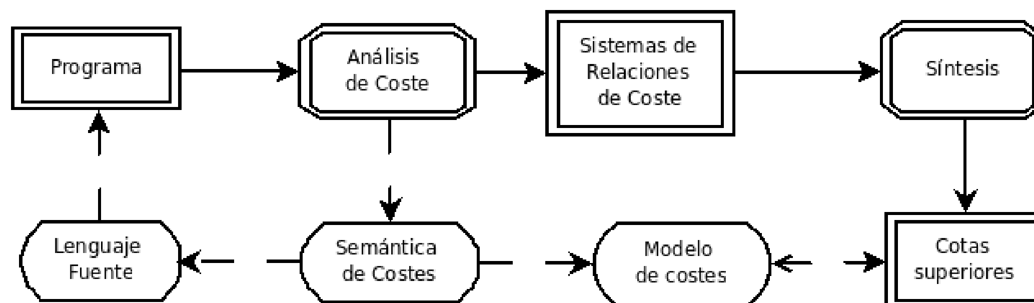
Los Sistemas de Relaciones de Coste (SRC), descritos en el segundo capítulo, cumplen estos requisitos tal y como se detalla en [2], y por ello son la base de nuestro proyecto.

Algoritmos

Así pues, el estudio de costes incluye las siguientes fases:

1. Proponer para nuestro lenguaje *fuente* una *semántica* composicional que determine cómo construir el SRC de un programa. Esta semántica deberá ajustarse para reflejar fielmente los costes que vaya a modelar.
2. **Análisis** de la estructura de un programa con esa *semántica* para generar un SRC que modele sus costes.
3. **Síntesis** de información útil, generalmente no trivial, a partir de esos SRC. Esa información incluye cotas de coste.

Para estudiar una aplicación o una biblioteca de cierta magnitud no es viable un proceso manual. Necesitamos algoritmos y programas que automaticen este proceso, y en esto consiste este proyecto de grado.



Corrección e incompletitud

El análisis es siempre correcto: para todo programa en lenguaje fuente extrae un SRC con sus costes. Pero por ello las herramientas de síntesis que intenten deducir información *no trivial* siempre serán incompletas.

Un problema no trivial es saber si un programa termina en tiempo finito. Podemos analizar ese programa y obtener un SRC que modele cuántas instrucciones ejecuta, información y éste SRC es suficiente para conocer la terminación.

Ahora bien, desde hace décadas se sabe que el problema de la terminación no es computable, ningún algoritmo puede decidir si cualquier otro programa termina o no. Si la herramienta de síntesis va a decidir si el programa termina, entonces será incompleta. usualmente decidirá condiciones más fuertes que la terminación, pudiendo así asegurar o no asegurar la terminación.

Proyectos

El proyecto COSTA [4] desarrolló un programa de análisis. El lenguaje fuente que utiliza es el JAVA BYTECODE, elemento clave de la ubicua plataforma JAVA, del cual extrae los costes del tiempo, empleando las técnicas vistas en [10], o los costes de memoria siguiendo lo que se indica en [5]

Especialmente notorias es la resolución del indeterminismo que genera la programación orientada a objetos o el manejo de excepciones JAVA. Además de lo que se puede leer en [3], con COSTA se están analizando bibliotecas y componentes como la JAVA COLLECTION FRAMEWORK o SWING, presentes en muchas aplicaciones.

Este proyecto de grado, por su parte, desarrolla una herramienta que procesa los SRC para sintetizar cotas superiores de coste. Para ello se basa en su predecesor, PUBS [3]. para desarrollar una implementación de segunda generación, más escalable y portable, de los conceptos y algoritmos descritos en [1].

[3] `http://www.clip.dia.fi.upm.es/Systems/PUBS/pubs.php`

1.3.3. SRC: Independencia del lenguaje

Tal y como se detalla en [2], los SRC no dependen de ningún lenguaje o paradigma de programación porque su especificación es puramente algebraica. Un SRC define relaciones numéricas con conceptos del álgebra lineal: expresiones lineales, sistemas de inecuaciones, números reales, etc.
Con ellos podemos modelar los costes de ejecución de un programa si abstraemos las características clave de ese lenguaje.

Necesitamos tratar de manera uniforme los datos del programa, ya estén estructurados como arrays, istas enlazadas o archivos. Para ello empleamos unas medidas numéricas, el **tamaño**.

- Una medida para una lista puede ser su longitud.
- Un árbol puede medirse por su altura o por el número de nodos.
- La magnitud de una tabla es la cantidad de entradas que contiene.

Asimismo, necesitamos representar en el SRC las instrucciones de control del flujo. Para ello se asocia una relación a cada bloque secuencial y los saltos se reflejan como llamadas entre esas relaciones.

Por último, para mostrar cómo cambian los datos durante la ejecución acompañamos las llamadas *de control* con inecuaciones lineales que muestran cómo cambia el tamaño.

```
void del(List l, int p, int a[], int la, int b[], int lb){
   while (l!=null) {
      if (l.data<p) {
         la=rm_vec(l.data, a, la);
      } else {
          lb=rm_vec(l.data, b, lb);
      }
      l=l.next;
   }
}
int rm_vec(int e, int a[], int la){
   int i=0;
   while (i<la && a[i]<e)  i++;
   for (int j=i; j<la−1; j++) a[j]=a[j+1];
   return la−1;
}
```

(1) $Del(l, a, la, b, lb) = 1 + C(l, a, la, b, lb)$
$\{b \geq lb, lb \geq 0, a \geq la, la \geq 0, l \geq 0\}$
(2) $C(l, a, la, b, lb) = 2$ $\{a \geq la, b \geq lb, b \geq 0, a \geq 0, l = 0\}$
(3) $C(l, a, la, b, lb) =$
$25 + D(a, la, 0) + E(la, j) + C(l', a, la-1, b, lb)$
$\{a \geq 0, a \geq la, b \geq lb, j \geq 0, b \geq 0, l > l', l > 0\}$
(4) $C(l, a, la, b, lb) =$
$24 + D(b, lb, 0) + E(lb, j) + C(l', a, la, b, lb-1)$
$\{b \geq 0, b \geq lb, a \geq la, j \geq 0, a \geq 0, l > l', l > 0\}$
(5) $D(a, la, i) = 3$ $\{i \geq la, a \geq la, i \geq 0\}$
(6) $D(a, la, i) = 8$ $\{i < la, a \geq la, i \geq 0\}$
(7) $D(a, la, i) = 10 + D(a, la, i+1)$ $\{i < la, a \geq la, i \geq 0\}$
(8) $E(la, j) = 5$ $\{j \geq la-1, j \geq 0\}$
(9) $E(la, j) = 15 + E(la, j+1)$ $\{j < la-1, j \geq 0\}$

Figura 1.1: Programa JAVA y SRC resultado del análisis.

Hay dos ventajas principales en emplear este proceso para el análisis:

- La estructura de los datos es transparente para los resultados: obtendremos el mismo coste para una búsqueda secuencial aunque esta sea sobre un array, una lista enlazada o una tabla.
- El lenguaje de implementación es transparente para el estudio de la eficiencia. Como muestra el siguiente ejemplo, para varios *programas* que implementen un mismo *algoritmo* obtendremos un mismo SRC.

Prolog

```
merge(This,[ ],R):- !,R = This.
merge([D|Next],[OD|ONext],R):-
   D>OD, !, R = [OD|T],
   merge([D|Next],ONext,T).
merge([D],O,R):- !,R = [D|O].
merge([D|Next],O,R):-
   R = [D|T], merge(Next,O,T).
```

Haskell

```
merge this [ ] = this
merge (d:next) o =
  if (d> od) then
    (od: merge (d:next) onext)
  else if (next==[ ]) then (d:o)
       else (d:merge next o)
  where (od:onext) = o
```

Java (recursive)

```
public class MLRec {
private int d;
private MLRec next;
public MLRec(int d, MLRec next){
  this.d = d;
  this.next = next;
}
public MLRec merge(MLRec o) {
  if (o == null) return this;      (1)m
  else if (d>o.d)
   return new MLRec(o.d,merge(o.next)); (2)m
  else if (next = null)
   return new MLRec(d,o);       (3)m
  else return new MLRec(d,next.merge(o)); (4)m
}
```

Figura 1.2: Implementaciones de merge en Prolog, Haskell y Java

$(1)_m$ $merge(this, o)=k_1$	$\{this\geq 1, o=0\}$
$(2)_m$ $merge(this, o)=k_3+merge(this, o')$	$\{this\geq 1, o\geq 1, o>o', o'\geq 0\}$
$(3)_m$ $merge(this, o)=k_2$	$\{this\geq 1, o\geq 1\}$
$(4)_m$ $merge(this, o)=k_4+merge(this', o)$	$\{this>this', this\geq 2, this'\geq 1, o\geq 1\}$
$\mathcal{M}_{\mathsf{ninst}}$	$k_1=4, k_2=26, k_3=26, k_4=29$

Figura 1.3: El mismo SRC para las tres implementaciones de merge.

1.4. SRC vs SED

Un **Sistemas de Ecuaciones en Diferencias (SED)** define recursivamente sucesiones numéricas. Un ejemplo típico es la sucesión de Fibonacci:

$$fib(X) \quad = \quad \begin{cases} 1 & \leftarrow X < 2 \\ fib(X-1) + fib(X-2) & \leftarrow X \geq 2 \end{cases}$$

Los SED son muy empleados en ciencia e ingeniería, y aparecen frecuentemente en el estudio de los sistemas caóticos, como la ecuación logística $x_{n+1} = rx_n(1 - x_n)$ o la ecuación cuadrática compleja $x_{n+1} = x_n^2 + c$ que define el fractal de Mandelbrot.

A menudo interesa resolver un SED, esto es definir sus sucesiones con ecuaciones explícitas no recursivas. Así, al resolver la sucesión de Fibonacci se obtiene esta ecuación:

$$fib(X) \quad = \quad \frac{\varphi^n - \frac{-1}{\varphi}^n}{\sqrt{5}} \text{ donde } \varphi = \frac{1+\sqrt{5}}{2}$$

La similitud entre los SED y las ecuaciones diferenciales (se considera a los SED como una versión discreta) ha permitido proporcionar algoritmos para resolver SED Las aplicaciones CAS (*Computer Algebra System*) comerciales, como Maple y Mathemathica, implementan estos algoritmos.

Los SRC y los SED pueden parecer muy similares a primera vista. Y podríamos pensar que un SRC solo es un tipo especial de SED, o que los mencionados CAS sirven para resolver un SRC, o incluso que los SED son más idóneos para el estudio de costes. Ninguna de las tres es acertada.

Primero debemos aclarar la relación entre SRC y SED:

- Hay sucesiones, como la de Fibonacci, que pueden modelarse tanto en SRC como en SED.
- Hay sucesiones que se pueden modelar en un SED pero no en un SRC.

$$X! \quad = \quad \begin{cases} 1 & \leftarrow X \leq 1 \\ X \times (X-1)! & \leftarrow X \geq 1 \end{cases}$$

- Y hay SRC cuya semántica excede a un SED.

$$coin = 0 \qquad\qquad coin = 1$$

Esta divergencia entre SED y SRC se debe al distinto modelo semántico de los SRC, el cual se escoge para modelar mejor el coste de los programas.

1.4.1. Indeterminismo

Un SED define **funciones**. $fib(0)$ solo vale 0 y $fib(5)$ solo vale 5. Sus ecuaciones son:

- mutuamente excluyentes, pues solo una es aplicable.
- deterministas, pues devuelven a lo sumo un valor.

Un SRC define **relaciones**. Sus ecuaciones son:

- no deterministas, pudiendo devolver varios valores cada una.
- no mutuamente excluyentes, pudiendo aplicarse varias para un mismo valor del parámetro de entrada.

Usamos SRC porque hay construcciones de programación que pueden volver la ejecución de un programa indeterminista. A continuación señalamos esas construcciones, con algunos ejemplos y su coste (caracteres impresos).

Condicionales

En una instrucción if-then-else, se ejecuta la rama then o la rama else, pero casi nunca podemos predecir cuál. Así pues, el coste de la instrucción condicional puede ser el de cualquier rama.

```
void saludo(boolean b){
  if(b) print("Hola'');
  else print(''Adiós'');
}
```

$$saludo = 4 \text{ //si imprime Hola}$$
$$saludo = 5 \text{ //si imprime Adiós}$$

Datos pseudoaleatorios

Este tipo de datos son necesarias en algunas aplicaciones, como generar claves criptográficas o simular procesos de azar. También es parte esencial de algoritmos como los del tipo Monte Carlo y otros.

```
int r = randomInt(10);
for(int i=0;i<r;i++){
  print("Ola'');
}
```

$$ola = 3X \leftarrow 0 \leq X < 10$$

Polimorfismo

En un lenguaje orientado a objetos, como Java o C++, se puede definir mediante herencia una jerarquía de clases, donde una clase general (Base) es especializada por otras clases (Derivadas). La clase base puede definir métodos abstractos que cada clase derivada reimplementa de distinto modo.

```
abstract class Animal{
  abstract String habla();
}


class Gato extends Animal{
  String habla(){
    return "Miau'';
  }
}
```

```
class Ave extends Animal{
  String habla(){
    return "Pio'';
  }
}
class Vaca extends Animal{
  String habla(){
    return "Muuuuuu'';
  }
}
```

El siguiente procedimiento muestra los dos elementos clave de la POO.

- Por el Principio de Sustitución, un objeto de una clase derivada puede estar sustituyendo a la clase base. Como ejemplo; la variable animal puede estar referida, indistintamente, a un perro, un ave o una vaca.
- Por el polimorfismo, la llamada al método habla() ejecuta una implementación que depende de la clase del objeto animal.

```
void hazHablar(Animal animal){
  print(animal.habla());
}
```

Como no podemos predecir el tipo concreto de un objeto, el coste de ejecutar un método polimórfico puede ser el de cualquier implementación.

$$
\begin{aligned}
hazHablar = 3 &\leftarrow \text{si es un ave} \\
hazHablar = 4 &\leftarrow \text{si es un gato} \\
hazHablar = 7 &\leftarrow \text{si es una vaca}
\end{aligned}
$$

Programación lógica

En Prolog un programa es un conjunto de predicados, definidos cada uno con varios hechos o reglas. Por ejemplo, el predicado saludo(X) se define con un hecho por cada posible valor de salida que queramos permitir.

```
saludo('Hola') .
saludo('Buenos días').
saludo('Buenas tardes').
saludo('Buenas noches').
```

```
saludaA(Nombre) :-
      saludo(X),
      concat(X, Persona, Y),
      print(Y).
```

En el predicado saludaA se llama al predicado saludo, empleando una variable lógica, X. El intérprete escogerá uno de los hechos que definen saludo para unificar la variable X con un literal. Como no podemos prever qué hecho escogerá, el coste de saludaA puede ser el que corresponde a cualquier saludo.

$$saludaA(X) = X + 4 \qquad saludaA(X) = X + 11$$
$$saludaA(X) = X + 13 \qquad saludaA(X) = X + 14$$

Encadenamiento hacia adelante

Algo semejante pasa en los sistemas de razonamientos como Clips y Jess. Aquí el cómputo consiste en la ejecución de una regla cuando en la base de hechos se cumple su condición o antecedente. Pero en cualquier momento pueden ejecutarse varias reglas, sin que podamos predecir cuál.

```
defrule atiende(
  clienteEspera
  =>
  print('Bienvenido')
  clienteEspera = false
)
```

```
defrule despacha(
  clienteEspera
  =>
  print('Vuelva mañana')
  clienteEspera = false
)
```

$$f = 10 \quad \leftarrow \quad \text{le atiende}$$
$$f = 13 \quad \leftarrow \quad \text{le despacha}$$

1.4.2. Imprecisión

En un SED las llamadas recursivas emplean argumentos exactos, que están determinados con precisión por los parámetros de la función.

$$f(X) = 5 + f(X-1)$$
$$f(X) = 10 + f(\frac{X}{2})$$

Tal precisión no es posible cuando la variable es el tamaño de unos datos, cuyo decrecimiento solo puede restirngirse con inecuaciones lineales.

Un ejemplo es el coste de recorrer un árbol binario en función del número de nodos X, que es igual al coste de recorrer cada subárbol. De esos subárboles solo sabemos que sus tamaños (Y, Z) son menores que X y suman $X - 1$.

$$f(X) = \begin{cases} c & \leftarrow X \leq 1 \\ c + f(Y) + f(Z) & \leftarrow X \geq 2, Y, Z < X, X = Y + Z + 1 \end{cases}$$

1.4.3. Bucles de múltiples argumentos

Un SRC puede incluir llamadas recursivas con múltiples argumentos, donde unos parámetros decrecen u otros crecen. Este tipo de bucles excede el ámbito de los SED y por ello la mencionadas aplicaciones de álgebra no pueden manejarlos.

$$\begin{aligned} f(A, B, C) &= & 3 \leftarrow B = C \\ f(A, B, C) &= & 5 + f(A-1, B-A, D) \leftarrow A \geq B, D < C \\ f(A, B, C) &= & 7 + f(\frac{A-B}{2}, C, C) \leftarrow A \geq B, C \geq 2B \end{aligned}$$

Y a la inversa, los SED pueden incluir bucles y expresiones que en cambio nunca aparecen en un SRC. Por ejemplo, las llamadas recursivas pueden incluir expresiones exponenciales o coeficientes polinómicos.

$$\begin{aligned} f(X) &= X + X^2 * f(\frac{X}{2}) \\ f(X) &= 1 + f(\sqrt{X}) \end{aligned}$$

Capítulo 2

Sistemas de Relaciones de Coste

2.1. Introducción intuitiva

2.1.1. Ecuaciones y Relaciones

Una **Relación de Coste (RC)** la definen varias **Ecuaciones de Coste (EC)**, cada una de las cuales defune una parte de su RC.

Como primer ejemplo, $f = 0$ significa que f *puede valer* 0, y a su vez

$$f = 0 \qquad\qquad f = 1$$

significa que f *puede valer* 0 y 1. Hay que remarcar que las EC no se excluyen entre sí. Por ello los elementos de esta RC *pueden ser* todos los que estén definidos por alguna ecuación.

Las RC suelen tener algunos parámetros de entrada, como en el ejemplo:

$$f(X) = 3 \qquad\qquad f(X) = nat(X)$$

Definamos algunos conceptos sobre ecuaciones y relaciones.

- Las *RC* se identifican por su nombre y número de parámetros.
- En una *EC*, a la izquierda de = está la **cabecera**, que denota su relación, y a la derecha el **cuerpo**, que define posibles valores.
- Un SRC es un conjunto de RC. Dos SRC son **equivalentes** si definen las mismas relaciones.

2.1.2. Ecuaciones y restricciones.

Las EC incluyen un sistema de ecuaciones e inecuaciones entre sus variabes que restringen los posibles valores. Así ocurre en la EC

$$f(X) = nat(X) \leftarrow 0 \leq X \leq 10$$

Esta ecuación indica que la RC $f(X)$ puede devolver cualquier número entre 0 y 10, valores de x que cumplen la restricción.

Algunos conceptos de ecuaciones y restricciones:

- Una **restricción redundante** es la que no modifica los valores de la ecuación. Por ejemplo, en $f = 5 \leftarrow X \leq 10$ la restricción no limita los resultados ni los parámetros, por cuanto es redundante.
- Una **ecuación** es **satisfactible** si sus restricciones pueden cumplirse en algún valor de variables y sino es **insatisfactible**. Así $f = 0 \leftarrow 0 = 1$ es insatisfactible y, como no define ningún valor, puede suprimirse.
- Un valor de los parámetros **encaja** *(matches)* en una ecuación si ésta es satisfactible para aquellos. Algunos parámetros podrán encajar en varias ecuaciones (indeterminismo) y otros en ninguna.
- El **dominio** de una ecuación son los vectores parámetro que encajan con ella. El dominio (rango) de una relación es la unión de los dominios (rangos) de sus ecuaciones.
- Una RC es **funcional** si para cada elemento de su dominio solo tiene un posible resultado.

2.1.3. Llamadas y recursión

Una EC de una relación puede incluir **llamadas** a otras RC. Así, en

$$\begin{array}{rcccc} f(X) & = & 3 + g(X) & \leftarrow & X \geq 1 \\ g(X) & = & 2^{nat(x)} & & \end{array}$$

en la primera ecuación se llama a g con unos **parámetros**, que son expresiones lineales sobre las variables de la ecuación. Esta EC nos indica que $f(X)$ puede valer cualquier $c = 3 + d$ si d es un valor de $g(X)$.

Existen varios conceptos asociados a las llamadas:

- Una llamada es **satisfactible** si sus parámetros forman parte del dominio de la relación llamada, y de lo contrario es insatisfactible. Una llamada a una relación vacía es insatisfactible.
- Para que unos parámetros encajen con la ecuación es necesario que las variables puedan tomar un valor en que satisfazcan todas las llamadas.
- Una EC es **cerrada** si no tiene llamadas. Una RC es cerrada si todas sus EC lo son y un SRC es cerrado si todas sus RC lo son.

A menudo interesa suprimir las llamadas de una EC, **desplegando *(unfolding)*** las EC que definen una RC. Por ejemplo en el SRC

$$\begin{array}{rcccl} f(X) & = & 3+g(X) & & \\ g(X) & = & 1 & \leftarrow & X \leq 15 \\ g(X) & = & 2 & \leftarrow & X \geq 15 \end{array}$$

se puede desplegar g en f para obtener este SRC

$$\begin{array}{rcccl} f(X) & = & 4 & \leftarrow & X \leq 15 \\ f(X) & = & 5 & \leftarrow & X \geq 15 \\ g(X) & = & 1 & \leftarrow & X \leq 15 \\ g(X) & = & 2 & \leftarrow & X \geq 15 \end{array}$$

Más complicación que las llamadas a otras RC traen las llamadas a la misma RC, las **llamadas recursivas** o **bucles**.

$$\begin{array}{rccccl} f(X) & = & 4 & & & \\ f(X) & = & 5 & +f(X-1) & \leftarrow & X \geq 1 \end{array}$$

Un gran problema de la recursión son los bucles infinitos, que pueden prolongarse indefinidamente, lo que genera ecuaciones que no devuelven valor. Así sucede en este SRC, donde f no devuelve ningún valor.

$$f(X) = 0 + f(X)$$

- Una RC es recursiva si tiene, directa o indirectamente, un bucle.
- Una EC es directamente recursiva si todas sus llamadas son bucles. Una RC es directamente recursiva si todas sus EC son cerradas o d.r.

2.2. Definición formal

Este apartado define formalmente todos los conceptos de un SRC.

2.2.1. Preliminares

Símbolos y notación

- $X, Y, \ldots$ representan variables de tipo racional. $\overline{X}^n$ es un vector.
- $a, b, \ldots$ representan números racionales. $\overline{a}^n$ representa un vector.
- $n, m, \ldots$ representan números naturales.
- $T, S, \ldots$ representan términos.
- φ, Ψ, ψ representan fórmulas numéricas.
- $[X/T]$ representa la **sustitución** de la variable X por el término T.
  $[S/T]$ representa la sustitución del término S por el término T.

Expresiones Lineales

Una **expresión lineal (EL)** *(Linear Expression)* es un término

$$a_0 * X_0 + a_0 * X_1 + \ldots + a_{n-1} * X_{n-1} + K \text{ donde}$$

- a_i son números racionales[1] no nulos, los **coeficientes**.
- X_i son **variables** que de tipo racional.
- K es un número racional, llamado **constante** o término independiente.
- $n \geq 0$ es el número de variables o **dimensión** $dim(exp)$.
  Una EL es constante si su dimensión es cero, esto es si no tiene variables.

A veces es más cómodo escribir una expresión lineal en forma vectorial

$$\overline{a} \cdot \overline{X} + K \text{ donde } \overline{a} = (a_0, \ldots, a_n - 1) \text{ y } \overline{X} = (X_0, \ldots, X_{n-1})$$

[1] Es discutible si podrían ser también reales o complejos.

Restricciones lineales.

Una **restricción lineal(RL)** *(Linear Constraint)* es una fórmula numérica φ de alguna de estas formas:

- Una igualdad $exp = 0$.
- Una desigualdad $exp \geq 0$
- Una desigualdad estricta $exp > 0$

donde exp es una EL $a_0 * X_0 + a_0 * X_1 + \ldots + a_n * X_n + K$ tal que

- $dim(exp) > 0$, hay por lo menos una variable libre.
- $|a_0| = 1$, y en el caso de una igualdad $a_0 = 1$.
- Una **solución** de φ es un vector $\bar{b}$ tal que $\varphi[\overline{X}/\bar{b}]$ es cierto. Definimos $sol(\varphi)$ como el conjunto de soluciones de φ:

$$sol(\varphi) = \{\bar{b} \in \mathbb{Q}^n | \varphi[\overline{X}/\bar{b}]\}$$

- Una RL φ es **satisfactible** si $\exists \bar{b}.\varphi[\overline{X}/\bar{b}]$, esto es si admite solución. De lo contrario es **insatisfactible**.

Relación de tamaño

Una **relación de tamaño (RT)** *(Size Relation)* Ψ es un sistema de $m \geq 0$ restricciones lineales $\varphi_1, \ldots, \varphi_m$, que representa su conjunción.

$$\begin{array}{rcc} a_{1,0} * X_{1,0} + \ldots + a_{m,n} * X_{1,n} + K_1 & R_1 & 0 \\ a_{2,0} * X_{2,0} + \ldots + a_{m,n} * X_{2,n} + K_2 & R_2 & 0 \\ \ldots & \ldots & \ldots \\ a_{m,0} * X_{m,0} + \ldots + a_{m,n} * X_{m,n} + K_m & R_m & 0 \end{array}$$

A veces se emplea la forma matricial de una RT

$$A \times XR0 \text{ donde}$$

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \text{ y } X = \begin{pmatrix} X_1 \\ X_2 \\ \ldots \\ X_n \end{pmatrix} \text{ y } R = \begin{pmatrix} R_1 \\ R_2 \\ \ldots \\ R_n \end{pmatrix}$$

- La **dimensión** de una RT es su cantidad de variables.
- Una **solución** de Ψ es un vector $\bar{b}$ solución de todas las φ_i.

$$sol(\Psi) = \bigcap_{i=1}^{m} sol(\varphi_i)$$

- Una RT es **satisfactible** si tiene solución, sino es insatisfactible.

El poliedro solución de una RT de dimensión n es el conjunto de puntos del espacio $\mathbb{Q}^n$ que son solución de la RT. Así, toda RT puede representarse geométricamente con su poliedro solución.

- Si una RT es insatisfactible, su PS es el poliedro vacío.
- Si una RT tiene alguna desigualdad estricta no redundante, entonces su PS estará abierto. De lo contrario será un poliedro cerrado.

La proyección de una RL es el proceso de eliminar algunas variables de la RT manteniendo las restricciones entre las demás.

$$\exists\ \overline{X}^m.\Psi(\overline{X}^m, \overline{Y}^n) = \Phi(\overline{X}^m) \Longleftrightarrow$$
$$sol(\Phi(\overline{X}^m)) = \{\bar{b} \in \mathbb{R}^m | \exists \overline{Y} \in \mathbb{R}^n).\Psi(\bar{b}, \overline{Y}^n)\}$$

Geométricamente, la proyección es el análogo de obtener una *sombra* del poliedro $(n+m)$-dimensional en un espacio n-dimensional.

Operador convexo de Hull $\sqcup$ es una operación que toma dos poliedros de dimensión n y obtiene el mínimo poliedro convexo que contiene a ambos.

El operador *Widening* toma varios poliedros e intenta deducir una progresión entre ellos y obtener así un poliedro **envolvente**. El *widening* nos permite generalizar varias RT obtenidas de manera secuencial. Por ejemplo,

$$\begin{array}{rcl} B &=& A-1 \\ B &\geq& 0 \end{array} \quad \text{y} \quad \begin{array}{rcl} B &=& A-2 \\ B &\geq& 0 \end{array}$$

se generalizan mediante *widening* con

$$\begin{array}{rcl} B &\leq& A \\ B &\geq& 0 \end{array}$$

2.2.2. Expresiones de Coste

Definición

El conjunto de **Expresiones de Coste (ExC)** *(Cost Expressions)* es el mínimo conjunto de términos que pueden formarse mediante aplicación finita de estas reglas:

- Todo **número** (Number) real $d \geq 0$ es una ExC.
- Si l es una expresión lineal no constante, entonces $nat(l)$ es una ExC, llamada Nat, que significa $nat(l) = max(0, l)$.
- Si $e_1, e_2, e_3, \ldots, e_m$ son ExC, con $m \geq 2$ y finito, entonces:
  - $e_1 + e_2 + e_3 + \ldots + e_m$ es una ExC **suma** (Sum), donde e_i son sus sumandos *(addends)*.
  - $e_1 * e_2 * e_3 * \ldots * e_m$ es una ExC **producto** (Product), donde e_i son sus factores *(factors)*.
  - $max(e_1, e_2, e_3, \ldots, e_m)$ es una ExC **máximo** (Max), donde los e_i son sus argumentos *(arguments)*.
  - $min(e_1, e_2, e_3, \ldots, e_m)$ es una ExC **mínimo** (Min), donde los e_i son sus argumentos *(arguments)*.
  - $e_1^{e_2}$ es una ExC **potencia** (Power), e_1 es base y e_2 exponente.
- Si a es una ExC cuyo valor mínimo es v y $d > 0$ es un número menor que v, entonces $a - d$ es una ExC llamada **resta** (Subtract).
- Su a es una ExC de valor mínimo $a \geq 1$ y $n \geq 2$ es un natural, entonces $log_n(a)$ es una ExC llamada **logaritmo** (Log).

Sea e una ExC. Se define $vars(e)$ como el conjunto de variables de e, esto es las variables que aparecen en sus componentes nat.

Definición 1. Sean e una ExC, $\overline{X}^n = vars(e)$ y $\overline{b} \in \mathbb{Q}^n$. La **evaluación** de e con $\overline{b}$ es igual a $v \in \mathbb{R}^+$ si $e[\overline{X}/\overline{b}] = v$.

Definición 2. Sean las Expresiones de Coste e,f con $vars(e) \subseteq vars(f) = \overline{X}^m$. Se define la igualdad entre ExC como

$$e = f \iff \forall(\overline{a}) \in \mathbb{Q}^m : e[\overline{X}/\overline{a}] = f[\overline{X}/\overline{a}]$$

es decir, dos ExC son iguales si dan el mismo valor en cualquier evaluación.

Orden, Valor mínimo y Monotonía

Definición 3. Orden entre expresiones Sean e, f dos expresiones de coste con $vars(e) = \overline{X}^m$ y $vars(f) = \overline{Y}^n$, y sea $\Psi(\overline{X}, \overline{Y})$ una RT. Se define la relación $e \sqsubseteq_\Psi f$, leida e *acota inferiormente a* f bajo Ψ, como

$$e \sqsubseteq_\Psi f \Longleftrightarrow \forall(\overline{a}, \overline{b}) \in \mathbb{R}^{m+n}.\Psi(\overline{a}, \overline{b}) : e[\overline{X}/\overline{a}] \leq f[\overline{Y}/\overline{b}]$$

es decir que en cualquier sustitución de las variables de e y f que satisfaga la restricción Ψ sucederá que el valor de e es menor o igual que el valor de f.

Si e es cota inferior de f bajo cualquier RT, se escribe $e \sqsubseteq f$

Ejemplos:

- Si $a, b \in \mathbb{R}^+$ y $a \leq b$ entonces $a \sqsubseteq b$.
- Sea $\Psi \equiv X \leq Y$, entonces $nat(X) \sqsubseteq_\Psi nat(Y)$
- Para cualquier ExC e, $log_n e \sqsubseteq e$ y $e - d \sqsubseteq e$.
- Para cualquier par de ExC e, f: $e \sqsubseteq e + f$, $e \sqsubseteq max(e, f)$, $min(e, f) \sqsubseteq e$

Teorema 1. La relación $\sqsubseteq$ define un orden parcial entre las ExC.

Por tanto, $(ExC, \sqsubseteq)$ es un conjunto parcialmente ordenado con un elemento mínimo, la expresión 0.

Definición 4. Para toda expresión de coste e existe un número real $v \in \mathbb{R}^+$, llamado **valor mínimo**, tal que $v \sqsubseteq e$. Este valor se obtiene sustituyende toda subexpresión *nat* por 0.

Definición 5. Principio de Monotonía de las ExC: Sean e,f dos expresiones de coste y sea Ψ una relación de tamaño. Para cualquier ExC g,

$$e \sqsubseteq_\Psi f \Rightarrow g \sqsubseteq_\Psi g[e/f]$$

es decir, al cambiar una componente e de g por otra expresión f que es cota superior de e se obtiene una expresión que es cota superior de la original.

El principio de Monotonía es fundamental en nuestro estudio, pues ello nos permite inferir composicionalmente las cotas superiores o inferiores de las ExC, y posteriormente de las relaciones de coste.

Forma normal

Nosotros queremos tener las ExC en forma normal, que se define como:

- Un número real está en forma normal.
- $nat(l)$ está en forma normal si l tiene dimensión no nula.
- Una suma está en forma normal si
  - Todos sus sumandos están en forma normal.
  - No hay dos sumandos iguales.
  - Ningún sumando es una suma.
  - Como máximo un sumando es un número, y éste no es cero.
- Un producto está en forma normal si
  - Todos sus factores están en forma normal.
  - No hay dos factores iguales.
  - Ningun factor es un producto.
  - Como máximo un factor es un número, y éste no es ni cero ni uno.
- Una operación max está en forma normal si
  - Todos sus argumentos están en forma normal.
  - No hay dos argumentos iguales.
  - Ningún argumento es un max.
  - Como máximo un argumento es un número.
- Una potencia está en forma normal si:
  - tanto su base como su exponente están en forma normal,
  - su base no es en sí misma una potencia.
  - al menos su base o su exponente no son un número
- Un logaritmo está en forma normal si su expresión está en forma normal y ésta no es ni un número ni un producto ni una potencia.
- Una resta está en FN si su expresión está en FN, no es un número y en caso de ser una suma entonces ningún sumando es un número.

2.2.3. Relaciones de Coste

Definición 6. Una **relación de coste (RC)** *(Cost Relation)* r de aridad finita $n \geq 0$ es una función

$$r : \mathbb{Q}^n \rightarrow P(\mathbb{R}^+)$$

que **puede ser definida** como la unión de varias ecuaciones de coste.

Se definen varios conceptos con las relaciones de coste.

- Para todo $n \in \ltimes$, $RC(n)$ denota el conjunto de relaciones de coste de aridad n. Una RC de aridad 0 es un **conjunto**.
- El **dominio** de una RC es el conjunto de vectores con al menos un valor asociado.
$$dom(r) = \{\bar{b} \in \mathbb{Q}^n | r(\bar{b}) \neq \phi\}$$
- Una RC es **total** si su dominio es $\mathbb{Q}^n$.
- Una RC es **funcional** si para cada vector de su dominio solo devuelve un valor: $\forall \bar{b} \in dom(r) : |r(\bar{b})| = 1$

Definición 7. Dos RC r,s de misma aridad n son **iguales** cuando

$$r = s \Longleftrightarrow \forall \bar{b} \in \mathbb{Q}^n : r(\bar{b}) = s(\bar{b})$$

esto es si definen los mismos valores para cualquier vector.

Definición 8. Sean r,s dos relaciones de coste de la misma aridad. r acota inferiormente a s ($r \sqsubseteq s$) si

1. $dom(s) \subseteq dom(r)$
2. Para cada vector $\bar{b} \in dom(s)$, $\forall (d, e) \in (r(\bar{b}), s(\bar{b})) : d \leq e$.

Respectivamente, r acota superiormente a s si $d \geq e$

Teorema 2. La relación $r \sqsubseteq s$ es una relación de orden parcial en el conjunto de las RC de aridad n, puesto que es **reflexiva**, **antisimétrica** y **transitiva**.

Por tanto, el conjunto $(RC(n), \sqsubseteq)$ es un orden parcial que tiene un elemento mínimo, la RC **constante cero**, $z^n(\overline{X}) = 0$.

2.2.4. Ecuación de coste

Definición

Una **Ecuación de Coste (EC)** *(Cost Equation)* es una fórmula con la siguiente estructura:

$$\overbrace{f(\underbrace{X_1, X_2, \ldots, X_n}_{\text{Parámetros}})}^{\text{Relación de Coste}} = \overbrace{exp + \underbrace{\sum_{i=1}^{m} g_i(p_{i,1}, \ldots, p_{i,s_i})}_{\text{Llamadas}}}^{\text{Cuerpo de la ecuación}} \leftarrow \overbrace{\Psi(\overline{X}, \overline{Y})}^{\text{Relación de tamaño}}$$

- exp ex una expresión de coste.
- φ es una relación de tamaño entre las variables de la ecuación.
- g_i son otras relaciones de coste.
- Los $p_{i,j}$, llamados argumentos, son expresiones lineales.

En una ecuación se distinguen dos tipos de variables:

- Los parámetros de la relación $\overline{X}^n$.
- Las que solo aparecen en el cuerpo y la RT, las variables libres $\overline{Y}^m$.

Forma normal

Se dice que una ecuación de coste está en forma normal si no hay una ecuación equivalente con menos variables libres. En concreto,

- La expresión y las llamadas están en forma normal.
- Todas las variables libres en Ψ aparecen en la expresión o las llamadas.
  $f(A) = 5 \leftarrow B > 0$ no está en f.n. Sí lo está $f(A) = 5$
- Ψ solo contiene igualdades entre los parámetros.
  $f(A) = g(A, B) \leftarrow B = 0$ no está en f.n, sí lo está $f(A) = g(A, 0)$

Llamadas

La ecuación de coste incluye varias llamadas con la forma

$$f(p_1, p_2, \ldots, p_l) \text{ donde}$$

- f es una relación de coste, la relación invocada.
- Los p_i son expresiones lineales sobre los parámetros y variables libres.

Definición 9. La **representación de una llamada** es un término

$$< f(\overline{X}) \to g(\overline{Y}), \Psi(\overline{X}, \overline{Y}) >$$

que representa que la RC f puede llamar a la relación g cuando se cumple la relación de tamaño Ψ entre sus parámetros.

Definición 10. Sean las ecuaciones de coste

$$f(\overline{X}) = exp_1(\overline{X}, \overline{Y}) + g(\overline{P}) + \sum_{i=1}^{k} h_i(r_{ij}) \leftarrow \Psi_1(\overline{X}, \overline{Y}) \qquad (2.1)$$

$$g(\overline{A}^n) = exp'_1(\overline{A}, \overline{B}) + \sum_{i=1}^{l} h_i(s_{ij}) \leftarrow \Psi_2(\overline{A}, \overline{B}) \qquad (2.2)$$

con $\overline{P} = (p_1, \ldots, p_n)$. Al **expandir** (*unfold*) la ecuación (2.2) en la llamada de la (2.1) se obtiene una nueva ecuación de coste

$$f(\overline{X}) = exp_1(\overline{X}, \overline{Y}) + exp'_1(\overline{P}, \overline{B}) + \sum_{i=1}^{k} h_i(r_{ij}) + \sum_{i=1}^{l} h_i(s_{ij}[\overline{A}/\overline{P}]) \leftarrow \Psi_1(\overline{X}, \overline{Y}) \wedge \Psi_2(\overline{P}, \overline{B})$$

Conceptos sobre la recursión

- Una llamada es **recursiva**, o se dice que es un **bucle**, si la relación invocada es la que se define en la ecuación.
- Una EC es recursiva si contiene algún bucle. Una RC es recursiva si alguna EC es recursiva.

Definición 11. La **representación de un bucle** es un término

$$< f(\overline{X}) \to f(\overline{Y}), \Psi(\overline{X}, \overline{Y}) >$$

que representa que la RC f contiene un bucle con unos ciertos parámetros.

Semántica de las EC

Definición 12. Sea E una ecuación de coste como la escrita arriba. Se dice que E define una subrelación de coste $f_E : \mathbb{Q}^n \rightarrow P(\mathbb{R}^+)$ como

$$\forall \overline{a} \in \mathbb{Q}^n : f_E(\overline{a}) = \{v \in \mathbb{R}^+ | \exists \overline{b} :$$
$$\Psi(\overline{a}, \overline{b}) \wedge v = exp[\overline{X}, \overline{Y}/\overline{a}, \overline{b}] + \sum_{i=1}^{m} d_i \wedge$$
$$\forall_{i=1}^{m} d_i \in g_i(p_{i,1}, \ldots, p_{i,s_i})[\overline{X}, \overline{Y}/\overline{a}, \overline{b}]\}$$

Esa *sencilla* ecuación de arriba indica que...

- para cualquier vector de entrada $\overline{b}$,
- $f(\overline{a})$ puede devolver cualquier valor v para el cual
- existe un vector $f(\overline{b})$, de valores para las variables libres, con el que
  - se cumple la Relación de tamaño Ψ,
  - al evaluar la expresión se obtiene e,
  - cada llamada g_i puede devolver un resultado d_i,
  - v se obtiene como suma de e y los d_i.

Definición 13. La **unión** de dos relaciones de coste f y g de aridad n es una relación de coste de h de aridad n definida como:

$$h = f \cup g \Longleftrightarrow \forall \overline{X} \in \mathbb{Q}^n : h(\overline{X}) = \{v \in \mathbb{R}^+ | v \in f(\overline{X}) \vee v \in g(\overline{X})\}$$

es decir, la relación que a cada vector de tamaño n devuelve los valores que definen las relaciones f y g.

Definición 14. Sean varias ecuaciones de coste $E_1, E_2, \ldots, E_m$que definen una misma RC. La **relación de coste definida** por esas ecuaciones es

$$f = f_{E_1} \cup f_{E_2} \cup \ldots \cup f_{E_N}$$

2.2.5. Sistemas de Relaciones de coste

Definición 15. Un **Sistema de Relaciones de Coste (SRC)** *(Cost Relation System CRS)* es un conjunto de EC que definen varias RC.

Sea un SRC S y sean f, g varias RC:

- $rel(S)$ denota el conjunto de RC definidas en S.
- $def(S, f)$ es el conjunto de EC que definen f. f es exactamente la RC definida entre las ecuaciones de $def(S, f)$.

$$f_S = \bigcup_{i=1}^{m} f_E | E \in def(S, f)$$

- $clients(S, f)$ es el conjunto de RC llamadas en las EC de $def(f)$.

En adelante supondremos que un SRC está autocontenido, que todas las llamadas en sus ecuaciones tienen que dirigirse a relaciones en $rel(S)$.

2.2.6. Árboles de evaluación

Una forma de representar la semántica de las llamadas a las RC de un SRC son los árboles de evaluación.

Definición 16. Sea un SRC S, sea f una relación de coste de aridad n y sea $\overline{a} \in \mathbb{Q}^n$. Un **árbol de evaluación** (AE) en S para $f(\overline{a})$ es cualquier árbol

$$nodo(f(\overline{a}), e, [T_1, T_2, \ldots, T_m]) \text{ donde}$$

de tal manera que existe alguna ecuación en $def(f)$

$$f(\overline{X}) = exp(\overline{X}, \overline{Y}) + \sum_{i=1}^{m} g_i \leftarrow \Psi(\overline{X}, \overline{Y})$$

cuyas variables libres son $\overline{Y}$, y para éstas hay algún valor $\overline{b}$ tal que

- Se satisface la relación de tamaño Ψ
- e es el resultado de evaluar exp, el **coste local**.
- cada T_i es a su vez un árbol de evaluación en S para la llamada g_i.

Se define $AE(f(\overline{X}))$ al conjunto de árboles de evaluación de f en S.

2.3. Cálculo de cotas superiores

La tarea a la que se dedica este proyecto, y especialmente este estudio sobre los Sistemas de Relaciones de Coste, es poder acotar las relaciones de coste de un SRC y, así, los costes de ejecución de un programa.

Dada la incompletitud señalada en el capítulo anterior, ningún algoritmo puede procesar satisfactoriamente todos los SRC. Aquí presentamos un proceso que, empero, cubre numerosos ejemplos muy frecuentes.

En concreto, queremos procesar un SRC S y obtener, para cada RC f en S, una nueva RC f' funcional y cerrada que es cota superior de f. Para ello nos valemos del concepto de árbol de evaluación.

2.3.1. Fórmula general

Grosso modo, se acotan los resultados de $f(\overline{X})$ empleando un AE imaginario para $f(\overline{X})$ que devuelve un coste mayor que todos los AE reales en S para esa llamada. Tal AE imaginario es más profundo que cualquier árbol real, tiene mayor coste local en las hojas y ramas.

Proposición 1. Sea el CRS S y sea f una RC directamente recursiva de S. La relación de coste cerrada y funcional f' definida como

$$\boxed{f'(\overline{X}) = int(\overline{X}) * rec(\overline{X}) + hoj(\overline{X}) * bas(\overline{X})}$$

- $int(\overline{X})$ es una cota superior del número de nodos internos (no hojas) de todos los árboles en $AE(f(\overline{X}))$.
- $rec(\overline{X})$ es una c.s. del coste local de los nodos internos de los árboles en $AE(f(\overline{X}))$.
- $hoj(\overline{X})$ es una c.s. del número de hojas de los árboles de en $AE(f(\overline{X}))$.
- $bas(\overline{X})$ es una c.s. del coste local de las hojas.

es una cota superior de f.

Otra definición alternativa es la siguiente:

$$\boxed{f'(\overline{X}) = \begin{cases} alt(\overline{X}) * rec(\overline{X}) + bas(\overline{X}) & \leftarrow r = 1 \\ \frac{r^{alt(\overline{X})}-1}{r-1} * rec(\overline{X}) + r^{alt(\overline{X})} * bas(\overline{X}) & \leftarrow r \geq 2 \end{cases}}$$

donde $alt(\overline{X})$ es una cota superior de la altura de los árboles de evaluación para $f(\overline{X})$, y r es el factor de ramificiación de estos AE.

2.3.2. Ranking functions

Como hemos visto en la fórmula anterior, necesitamos encontrar una cota superior de la altura de un árbol de evaluación. Ello lo logramos mediante el empleo de las *ranking function.*

Definición 17. Sea el bucle

$$L =< f(\overline{X}) \rightarrow f(\overline{X'}, \Psi(\overline{X}, \overline{X}') >$$

r es una ***ranking function*** de L es una expresión lineal r sobre las variables de $\overline{X}$ tal que $nat(r)$ es una cota superior del número de iteraciones de L. La existencia de una rf para un bucle garantiza su terminación.

Proposición 2. Sean $L_1, \ldots, L_n$ los bucles de una RC recursiva f,

$$< f(\overline{X}) \rightarrow f(\overline{X}'), \Psi_i(\overline{X}, \overline{X}') >$$

y sean $r_1(\overline{X}), \ldots, r_n(\overline{X})$ sus *ranking function.* Entonces, la altura de todo árbol de evaluación de $f(\overline{A})$ está acotada superiormente por la ExC

$$max(nat(r_1(\overline{A})), \ldots, nat(r_n(\overline{A})))$$

Bucle	*Ranking function*
$f(X) = f(X-1) \leftarrow X-1 \geq 0$	X
$f(X) = f(X-2) \leftarrow X-2 \geq 0$	$\frac{X}{2}$
$f(X,Y) = f(X,Y-1) \leftarrow X-1 \geq 0, Y-1 \geq 0$	Y
$f(X,Y) = f(X,Y+1) \leftarrow X \geq Y$	$X - Y$

Cuadro 2.1: Ejemplos de bucles y *ranking functions*

Proposición 3. Sea un bucle $L =< f(\overline{X}) \rightarrow f(\overline{X}'), \Psi_i(\overline{X}, \overline{X}') >$ y sea r una *ranking function* de este bucle

- Si $\Psi \models r(\overline{X}) - r(\overline{X}') \geq \delta$, entonces $nat(\frac{r(\overline{X})}{\delta})$ es una cota superior del número de iteraciones del bucle.
- Si $\Psi \models \frac{r(\overline{X})}{r(\overline{X}')} \geq k$, entonces $log_k(nat(r(\overline{X})+1)+1)$ es una cota superior de las iteraciones del bucle.

El algoritmo que nosotros utilizamos para calcular *ranking function* lineales fue expuesto por primera vez en [8], aunque nosotros lo hemos modificado para obtener RF lo más precisas posibles.

2.3.3. Invariantes

Necesitamos encontrar ahora $rec(\overline{X})$, el coste local de cada nodo recursivo. Para eso debemos maximizar la Expresión de Coste de las ecuaciones recursivas que puedan desplegarse.

En algunas ocasiones esas ExC serán solo números, fáciles de tratar. Pero frecuentemente contendrá alguna subexpresión *nat* que debemos maximizar. Esto se logra con los invariantes.

Definición 18. Sea un bucle $L =< f(\overline{X}) \rightarrow f(\overline{X}'), \Psi_i(\overline{X}, \overline{X}') >$ Un invariante $\Omega(\overline{X_0}, \overline{X})$ es una RT entre las variables de una llamada original y las variables de la ecuación desplegada en cada iteración. Es, pues, un conjunto de condiciones que se cumplen en todas las iteraciones.

Bucle	Invariantes
$f(X) = f(X-1) \leftarrow X-1 \geq 0$	$X \leq X_0$
$f(X) = f(X-2) \leftarrow X-2 \geq 0$	$X \leq X_0$
$f(X,Y) = f(X,Y-1) \leftarrow X-1 \geq 0, Y-1 \geq 0$	$Y \leq Y_0, X = X_0$
$f(X,Y) = f(X,Y+1) \leftarrow X \geq Y$	$Y \geq Y_0, X = X_0$

Cuadro 2.2: Ejemplos de bucles y *ranking functions*

Proposición 4. Sea E una ecuación de coste

$$f(\overline{X}) = exp(\overline{X}, \overline{Y}) + \ldots \leftarrow \Psi(\overline{X}, \overline{Y})$$

y sea el invariante $\Omega(\overline{X_0}, \overline{X})$ obtenido de la evaluación de $f(\overline{X_0})$. Si cada subexpresión $nat(r)$ de exp se substituye por $nat(s)$ donde

- r es una EL sobre las variables $\overline{X}, \overline{Y}$,
- s es una EL sobre las variables $\overline{X_0}$,
- Se cumple que $\Omega(\overline{X_0}, \overline{X}) \wedge \Psi(\overline{X}, \overline{Y}) \models s \geq r$

entonces la nueva expresión exp' es una cota superior de la anterior. Además, $exp'(\overline{X_0})$ una cota superior del coste local de cada nodo no terminal que use esta ecuación de un AE de $f(\overline{X_0})$.

Por tanto, al mazimizar las expresiones lineales de la expresión de coste de las ecuaciones empleando un invariante obtenemos el término $rec(\overline{X_0})$.

El algoritmo para calcular los invariantes puede consultarse en [9].

2.3.4. Evaluación parcial

El proceso anterior solo puede aplicarse a RC directamente recursivas. Para eso debemos transformar un SRC a forma directamente recursiva.

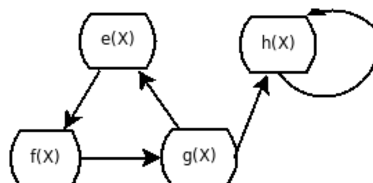
Análisis del grafo de llamadas

Definición 19. El **grafo de llamadas** de un SRC S es un grafo dirigido $< V, E >$ que

- Hay un nodo $v \in V$ por cada RC de S, y cada nodo solo representa una RC.
- Contiene una arista $(v, w) \in E$ por cada par (v, w) tal que la RC representada por v llama a la RC representada por w.
- No contiene ningún otro nodo ni arista.

Por ejemplo, veamos el grafo de llamadas de este SRC:

$$
\begin{aligned}
f(X) &= g(X) \\
g(X) &= h(X-1) + e(X) \\
h(X) &= 5+ \leftarrow X < 10 \\
h(X) &= 7+ \leftarrow X \geq 0 \\
e(X) &= 4 + f(X)
\end{aligned}
$$



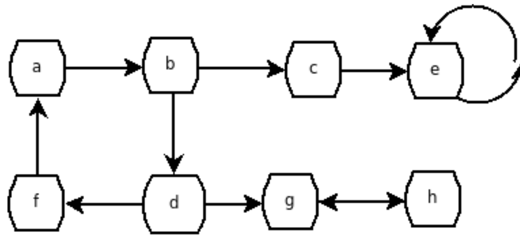
El grafo de llamadas es una buena forma de analizar un SRC.

- Una relación directamente recursiva aparece con un bucle, un arco que entra y sale del nodo.
- Una relación cerrada no tiene arcos salientes.

Definición 20. Una **componente fuertemente conexa** (CFC)) de un grafo dirigido $< V, E >$ es un subgrafo $< W, F >$ tal que entre dos nodos cualesquiera $v, w \in W$ existe un camino dentro de ese subgrafo.

Nuestro interés en las componentes fuertemente conexas radican en que cada CFF es una muestra de recursividad indirecta, la que se produce desde una relación a sí misma a través de llamadas a otras relaciones.

En el siguiente ejemplo, en el grafo se pueden observar las siguientes componentes fuertemente conexas: $\{a, b, d, f\}$, $\{g, h\}$,$\{e\}$. En el caso de un SRC, nos está indicando que e es una RC directamente recursiva o que existe un ciclo recursivo en a a través de b, f, g.



En general, la conectividad fuerte del grafo de llamadas es el equivalente gráfico de la recursividad directa o indirecta.

Nodos de cobertura

Para convertir un SRC a forma directamente recursiva necesitamos desplegar aquellas relaciones que no sean necesarias. Para saber qué ecuaciones suprimir y cuáles mantener empleamos los nodos de cobertura.

Definición 21. Sea $G =< V, E >$ una componente fuertemente conexa de un grafo dirigido. $v \in V$ es un **nodo de cobertura** (o punto de cobertura o *covering point*) de G si el subgrafo que queda al quitarle v a G no tiene ninguna componente fuertemente conexa.

Intuitivamente, un nodo de cobertura de una CFC es un nodo por el que pasan todos los ciclos de esa CFC. De tal manera que si se suprime el nodo de cobertura, se pierde la fuerte conectividad.

Para el grafo anterior estos son los puntos de cobertura:

Bucle	Invariantes
$f(X) = f(X-1) \leftarrow X-1 \geq 0$	$X \leq X_0$
$f(X) = f(X-2) \leftarrow X-2 \geq 0$	$X \leq X_0$
$f(X,Y) = f(X,Y-1) \leftarrow X-1 \geq 0, Y-1 \geq 0$	$Y \leq Y_0, X = X_0$
$f(X,Y) = f(X,Y+1) \leftarrow X \geq Y$	$Y \geq Y_0, X = X_0$

Cuadro 2.3: Ejemplos de bucles y *ranking functions*

Cuando se trata del grafo de llamadas de un SRC, ello implica la pérdida de la recursividad indirecta. Y una RC sin ciclos indirectos puede desplegarse (unfold) y suprimirse del sistema.

Proposición 5. Sea un SRC S y sea G su grafo de llamadas. Si toda componente fuertemente conexa de G tiene al menos un punto de cobertura, entonces S se puede reducir a forma directamente recursiva.

Por lo tanto, para escoger el conjunto de relaciones de coste que se despliegan y eliminan del sistema, se sigue este procedimiento:

1. Los nodos aislados (no están en ninguna CFC) que reciben alguna llamada pueden desplegarse del sistema.

2. Para cada CFC, se escoge uno de sus nodos de cobertura para no desplegarlo (relación residual) y se despliegan los demás nodos del CFC.

Al seguir este procedimiento, se obtiene un SRC directamente recursivo y sin CFC con más de un nodo.

Capítulo 3

Especificación de Requisitos

3.1. Sistemas y subsistemas

El componente de software, en adelante **Resolutor**, toma como entrada unos Sistemas de Relaciones de Coste, generados por las herramientas de análisis, y genera como salidas otros SRC con las cotas superiores inferidas.

La aplicación, en adelante **Prototipo**, expone la funcionalidad del Resolutor con una interfaz interactiva.

3.2. Requisitos Funcionales

3.2.1. Resolutor

El Resolutor debe:

1. Representar los CRS.
2. Ofrecer una vía uniforme de acceder a los datos de un SRC.
3. Interpretar *(parse)* un SRC escrito en texto.
4. Importar y exportar SRC a archivos de texto y a archivos binarios.
5. Transformar un SRC mediante recursión directa.
    a) Obtener el grafo de llamadas de un SRC.
    b) Encontrar sus componentes fuertemente conexas.

c) Encontrar los nodos de cobertura de cada CFC.

d) Obtener el conjunto de relaciones residuales.

6. Si fuere el caso, avisar de que esa transformación es imposible.

7. Calcular las Ranking Functions de una relación recursiva.

8. Calcular los invariantes de los bucles de una ecuación recursiva.

9. Obtener para una relación una cota superior en forma cerrada.

3.2.2. Prototipo

El prototipo permitirá al usuario:

1. Gestionar una *pizarra* con varios SRC.

2. Añadir un nuevo SRC a la pizarra.

3. Navegar por la pizarra y escoger entre los SRC disponibles.

4. Guardar un SRC en un archivo binario.

5. Mostrar las ecuaciones de un SRC.

6. Dibujar el grafo de llamadas de un SRC.

7. Ver las relaciones recursivas en el SRC.

8. Mostrar para cada relación recursiva sus bucles, y en cada bucle:

    - Decir si termina o no. Es decir, si se ha podido encontrar una *ranking function* lineal.
    - En caso de que terminara, mostrar esa *ranking function*.

9. Calcular y mostrar los invariantes de una relación recursiva.

10. Ejecutar la evaluación parcial sobre un SRC.

11. Calcular las cotas superiores de las relaciones de un SRC.

3.3. Requisitos no funcionales

3.3.1. R.N.F. de implementación

El resolutor y el prototipo forman parte de un proyecto internacional, y por tanto su implementación debe ajustarse a las políticas de este proyecto. Estas políticas están orientadas a facilitar la difusión de estas herramientas entre los programadores, ya sean los usuarios o los desarrolladores del sistema.

Lenguaje

El Resolutor y el Prototipo serán implementados en la última versión hasta la fecha del JAVA PROGRAMMING LANGUAGE.

Idioma

Por ser la *lingua franca* más común en ingeniería informática, el idioma del proyecto es el inglés. En consecuencia:

La nomenclatura del código fuente del **resolutor** y del **prototipo** estará en inglés y se basará en los conceptos de los SRC. Esta nomenclatura incluye las siguientes reglas de identificadores:

- *Packages* como partes principales del sistema.
- Elementos de un SRC como nombres de clases.
- Componentes de cada objeto como atributos.
- Operaciones y transformaciones en esos objetos como métodos.

La documentación del Resolutor y del Prototipo se escribirá en inglés. Esta documentación incluirá:

- Los JAVADOCS del API de las clases del resolutor.
- Los comentarios en código para los desarrolladores.
- Los diseños UML de la arquitectura del sistema.
- La documentación de usuario.

La interfaz gráfica de usuario del prototipo estará rotulada en inglés, en todos sus títulos, sus diálogos, sus menús y en la ayuda contextual.

Licencia y coste

El Resolutor y el Prototipo se distribuirán bajo alguna licencia de software libre, *open source* y gratuito. Esto implica que cualquier persona podrá:

- Obtener el código fuente o el código binario del programa.
- Compilar el código fuente,
- Ejecutar la aplicación **Prototipo**.
- Integrar o enlazar el resolutor o el prototipo en una aplicación mayor,
- Modificar libremente el Resolutor o el Prototipo

El formato en que se entregue el prototipo deberá ser un formato estándar o uno propio de alguna aplicación disponible como software libre y gratuito.

Se prohíbe por tanto que el sistema dependa de cualquier programa, biblioteca, herramienta o aplicacion que no sea software libre y que no se pueda obtener gratuitamente. Por ejemplo:

- Sistemas operativos como Microsoft Windows o Apple Mac OS.
- *Integrated Development Environments* como Borland JBuilder o IBM Rational Application Developer.

A modo de prueba, deberá ser posible compilar y ejecutar el programa en el sistema operativo Ubuntu.

Portabilidad

- El programa será desarrollado en el lenguaje de programación Java.
- Los componentes y bibliotecas empleados deberán estar dearrollados en un lenguaje portable (compilación independiente de cualquier Instruction Set Arquitecture) como C++ o preferiblemente también Java.

Modularidad

Para extenderse fácilmente en el futuro, el programa seguirá una arquitectura modular en la cual se distingan estos módulos como Java Packages.

3.3.2. Requisitos no funcionales del producto

Estándares de Calidad

Código En la implementación se seguirán las *Code Conventions for the Java Programming Language.*

Documentación El API del Resolutor debe documentarse ampliamente:

- En cada clase se detallará qué objeto de los SRC se modela, qué elementos lo componen y qué valores pueden tomar.
- Para cada atributo, su significado y las restricciones o invariantes sobre sus posibles valores.
- Para cada método del API, qué hace, qué valores acepta como parámetros y qué excepciones puede lanzar y en qué condiciones.

Pruebas Cada clase JAVA del **Resolutor** deberá contar con un amplio conjunto de casos de prueba, que comprobaráan su corrección y robustez.

Usabilidad

El **prototipo** tendrá una interfaz gráfica minimalista, que exponga de manera simple las funcionalidades que ofrece.

Eficiencia

Aunque no es un requisito prioritario, se deberán implementar los algoritmos de la manera más eficiente posible, tanto en el tiempo de proceso como en el consumo de memoria.

Robustez

El programa y sus módulos deberán garantizar ciertas condiciones internas. El prototipo deberá mostrar un buen manejo de excepciones.

Otros RnF

Dado que la aplicación no maneja ningún tipo de datos personales ni estiona recursos *hardware*, no hay ningún requisito de **seguridad** y **fiabilidad** directamente aplicables.

Capítulo 4

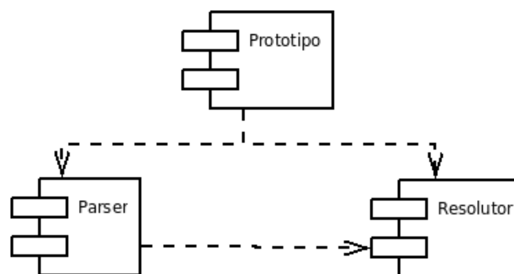
Desarrollo del sistema

4.1. Arquitectura del sistema

4.1.1. Arquitectura

Se distinguen tres componentes principales:

- El **Resolutor** contiene las clases que modelan y representan un SRC y sus elementos, e implementan todos los algoritmos necesarios para la obtención de cotas superiores.
- El **Parser** procesa archivos o cadenas de texto que describen un SRC y emplea los métodos de la clase Factory, del **Resolutor**, para construirlo.
- El **Prototipo** es una aplicación con interfaz gráfica para importar SRC de archivos de texto, interpretarlos con un Parser y mostrar qué se obtiene al aplicar los métodos del Resolutor.



Arquitectura de componentes del sistema

4.1.2. Diseño Detallado del Resolutor

El Resolutor ha sido construido con un diseño orientado a objetos, donde los complicados algoritmos para transformar elementos se integran exactamente en la clase que representa esos objetos.

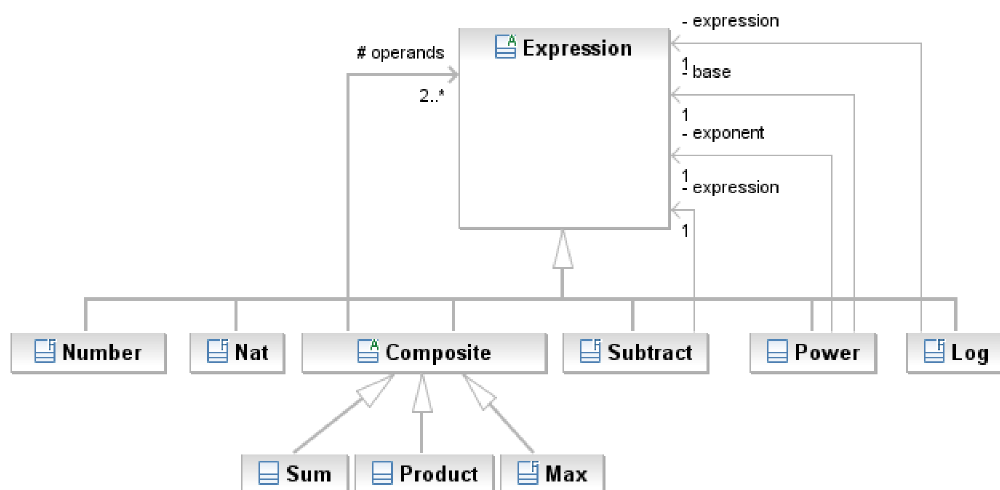
Alternativa Visitor

Un diseño alternativo, empleado inicialmente, proporcionarba una interfaz Visitable para acceder a los datos con varios Visitor, cada uno de los cuales implementaba un proceso del resolutor.

Así se habría logrado un Resolutor extensible, que integrare más algoritmos con más visitantes. Pero al dispersar el proceso en varios Visitor se perdía la cohesión de las clases de datos y se obtenía una implementación confusa e ineficiente. Por ello esta alternativa fue posteriormente descartada.

Expresiones de Coste

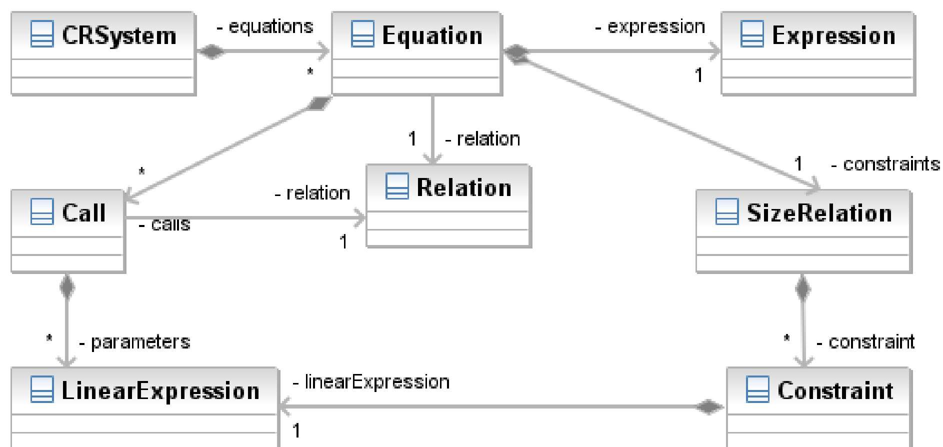
Las Expresiones de Coste: las clases que implementan las ExC se hallan en el JAVA *package* crs.expression. Se han diseñado empleando el patrón de diseño Composite, de tal manera que las ExC compuestas contienen una o varias ExC.



Elementos del SRC

Las otras clases importantes del Resolutor son las siguientes:

- CRSystem representa un SRC, y se ocupa de:
  - Ejecutar la evaluación parcial, obteniendo las relaciones residuales del sistema y desplegando unas ecuaciones en otras.
  - Calcular las cotas superiores de sus RC.
- Equation representa una EC. Implementa el proceso de desplegar una ecuación dentro de otra *(unfolding)* y maximizar las ExC con invariantes.
- SizeRelation representa una relación de tamaño, e implementa las operaciones de proyección, cierre convexo de Hull y *widening*.
- Relation es una *etiqueta* para distinguir las relaciones de coste, con su nombre y número de parámetros.
- Loop representa un bucle y calcula su *ranking function*.



Algunas notas sobre esas clases:

- Implementan el interfaz java.io.Serializable, para así almacenar sus datos en archivos binarios.
- Implementan el interfaz java.lang.Comparable para usar TreeSet como colección preferible. Así logramos una sola forma de expresar cada valor.

4.1.3. Diseño Detallado del parser

El intérprete ha sido construido en JavaCC, con la siguiente gramática. Esta gramática no detalla precedencia (usual) ni asociatividad (a izquierdas).

En notación, A^+ indica una o más repeticiones, A^* indica cero o más repeticiones, A? omisible, $A|B$ alternativa y parentizamos {A}. Se distinguen categorías *Sintactica*, **Lexica** y palabra `Reservada`.

$$
\begin{aligned}
Sistema &::= Ecuacion^+ \\
Ecuacion &::= \texttt{eq}(Header, Expression, Calls, Constraints)\cdot \\
Header &::= \textbf{FunID}\{(\textbf{VarID}\{,\textbf{VarID}\}^*)\}? \\
Expression &::= \textbf{Number} \\
&\quad|\quad \texttt{nat}(LinearExp) \\
&\quad|\quad Expression + Expression \\
&\quad|\quad Expression \times Expression \\
&\quad|\quad Expression - \textbf{Number} \\
&\quad|\quad Expression/\textbf{Number} \\
&\quad|\quad \texttt{pow}(Expression, Expression) \\
&\quad|\quad \texttt{log}(\textbf{Number}, Expression) \\
&\quad|\quad \texttt{max}([Expression\{,Expression\}^+]) \\
&\quad|\quad (Expression) \\
Calls &::= [\{Call\{,Call\}^*\}?] \\
Call &::= \textbf{FunID}\{(LinearExp\{,LinearExp\})\}? \\
Constraints &::= [\{Constraint\{,Constraint\}^*\}?] \\
Constraint &::= LinearExp\textbf{Relation}LinearExp \\
LinearExp &::= LinearElement\{\{+|-\}LinearElement\}^+ \\
LinearElement &::= Coefficient\{*\textbf{VarID}\}?|\{-\}?\textbf{VarID} \\
Coefficient &::= \{-\}?\textbf{Number}\{/\textbf{Number}\}?
\end{aligned}
$$

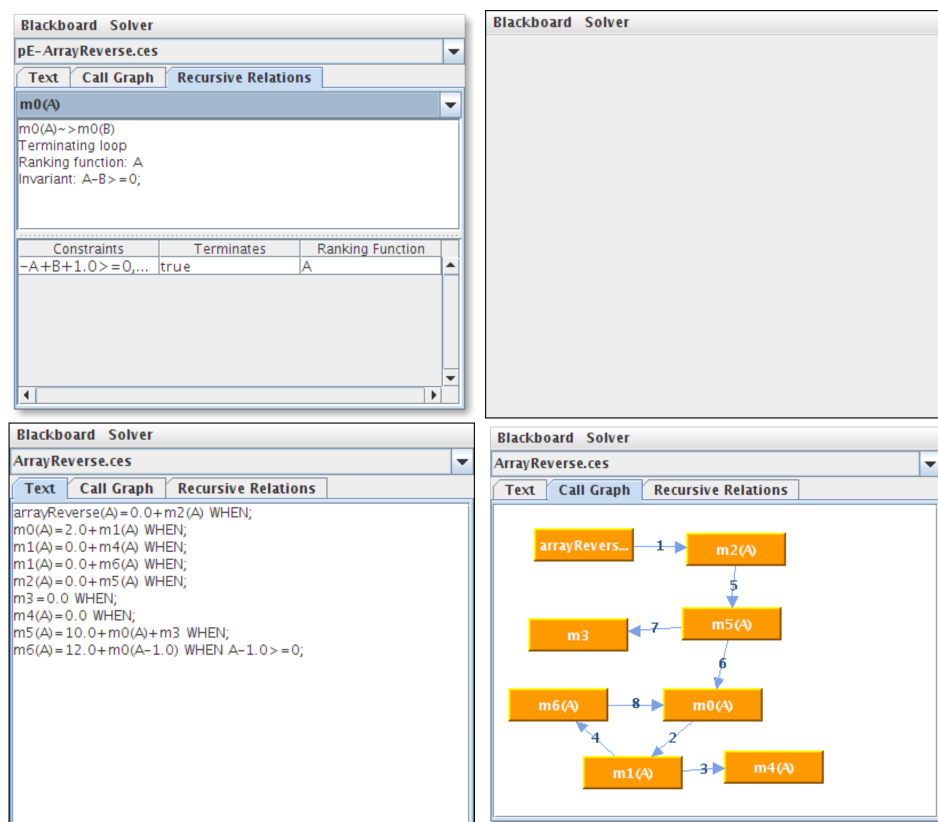
$$
\begin{aligned}
Relation &= \; < \; \leq \; = \; \geq \; > \\
VarID &= UpperCase, Letter^* \\
FunID &= LowerCase, Letter^*
\end{aligned}
$$

4.1.4. Diseño del prototipo

El Prototipo es una aplicación con una interfaz gráfica sencilla, que contiene estos elementos:

- Menú *Blackboard*, que permite cargar un SRC de un archivo, añadiéndolo a la pizarra, o limpiar esta pizarra.
- Menú *Solver*, que permite ejecutar la evaluación parcial de un SRC y el cálculo de cotas superiores.
- Selector desplegable, para escoger qué SRC se está observando.
- Pestaña de texto, para ver las ecuaciones de un SRC.
- Pestaña del grafo de llamadas.
- Pestaña de las relaciones recursivas, para ver las ranking functions de cada bucle y los invariantes de una relación recursiva.

Esta interfaz ha sido implementada en SWING, en aras de obtener una mayor portabilidad y estabilidad para el prototipo.



4.2. Herramientas usadas

4.2.1. JavaCC

Java Compiler Compiler[1], más conocido como JavaCC, es una herramienta que permite generar en Java compiladores descendentes predictivos recursivos. Toma una gramática en una notación pseudo-EBNF y genera un conjunto de clases Java que constituyen un parser del lenguaje deseado.

Actualmente JavaCC es la herramienta más usada para generar compiladores en Java, debido a que:

- Genera compiladores muy portables en Java.
- Está publicado bajo la licencia se software libre permisiva BSD.
- Los analizadores sintácticos que genera son muy eficientes.
- Mantienen un repositorio con el código JavaCC de varios lenguajes de programación, incluyendo Java, C++, Haskell, Prolog o VHDL.

En nuestro proyecto hemos usado JavaCC para implementar el intérprete de los SRC. Al ser una herramienta muy difundida y fácil de usar, creemos que JavaCC da una buena especificación sin comprometer la portabilidad de nuestro programa, facilita las futuras modificaciones y extensiones que puedan hacerse. Como los compiladores que genera son muy rápidos y eficientes, ello mejora la eficiencia de nuestro sistema.

Otra ventaja menor de JavaCC es que hemos podido reutilizar el código disponible en sus repositorios. Ese código incluía especificaciones de Prolog que y de generar compiladores rápidos, los mencionados repositorios nos permitieron reutilizar parte del código de los procesadores de Prolog.

Por ello, creemos que JavaCC es un magnífico e imprescindible complemento para el desarrollo de programas en Java, y desde aquí recomendamos su uso en proyectos académicos e industriales.

[1] https://javacc.dev.java.net

4.2.2. JUnit

JUnit[2] es un *framework* de pruebas unitarias para Java. Fue desarrollado por Kent Beck a principios de esta década como parte del desarrollo dirigido por pruebas *(Test Driven Development)*, tal y como se puede leer en [7].

JUnit nos permite programar unos métodos (comúnmente *junits*) que prueban intensivamente la interfaz de una clase. Con estos métodos podemos averiguar observando si sus métodos devuelven los resultados (o lanzan las excepciones) debidas.

Si JUnit es de gran ayuda en cualquier proyecto, con mayor razón en el nuestro. Al implementar las delicadas estructuras de datos y los algoritmos de álgebra es muy fácil cometer un ligero fallo que luego cuesta encontrar y depurar o que no aparece hasta más adelante.

Sin embargo, con ejecutar varios *junits* después de implementarlo, podemos detectar, encontrar y depurar el error que se haya cometido. Gracias a JUnit, podemos comprobar si esos métodos están correctamente programados y detectar y localizar rápidamente los fallos, ahorrándonos mucho esfuerzo de implementación.

Desde aquí recomendamos a todos los desarrolladores de Java a desarrollar *junits* para sus proyectos, aunque no sigan una metodología como *XP* o *Test Driven Development.* Ello les reportará dos grandes beneficios:

- Internamente, facilitará la depuración del programa y les dará una buena guía para refactorizar el programa.
- Externamente, ofrecer un conjunto de JUnits que prueben el API de una clase es una buena garantía de calidad.

[2] `http://www.junit.org`

4.3. Bibliotecas reutilizadas

4.3.1. PPL

La Parma Polyhedra Library (PPL) es una biblioteca de álgebra lineal para C++ desarrollada por Roberto Bagnara y otros miembros del *Computer Science Group* de la *Università degli Studi di Parma*[3].

Esta biblioteca fue desarrollada como resultado de las investigaciones que se reportan en [6], y se distribuye bajo la *GNU General Public License*.

Esta biblioteca resulta de gran utilidad en el Resolutor, pues algunos cálculos muy importantes como

- eliminar algunas variables de una ecuación (proyección).
- calcular la ranking function de un bucle
- maximizar una expresión lineal con un invariante

están implementados sobre PPL. Y esta biblioteca ejecuta esas tareas de manera muy eficiente y correcta.

Problemas de integración

Esta biblioteca nos trajo problemas al integrarla con el resto del sistema. Ello se debe a que PPL está programada en C++. Cuando compilamos PPL obtenemos un código binario nativo, dependiente de la *Instruction Set Arquitecture* del ordenador. Para usarla en Java es necesario emplear el *Java Native Interface*.

Las dificultades para compilar esa biblioteca, instalarla en el sistema y enlazarla desde la aplicación a través del JNI retrasaron varios meses el avance del proyecto dado que fue imposible implementar algunos algoritmos. Afortunadamente al final logramos integrarla gracias a la colaboración que aquí agradecemos a Samir Genaim, y a la ayuda de otros miembros de CLIP.

[3] http://www.cs.unipr.it/ppl/

Portabilidad

PPL ha lastrado la portabilidad del Resolutor debido a que presenta dos serias dependencias:

- El código fuente suele distribuirse para g++ *(GNU C++ Compiler)* y este compilador solo se halla disponible en distribuciones de LINUX.
- PPL depende a su vez de otra biblioteca, la GMP[4] que se integra mejor en las distribuciones LINUX.

En consecuencia, el prototipo solo puede ejecutarse en una máquina con sistema operativo Linux y que haya preinstalado la . Ello se aleja claramente de la portabilidad del código binario que proporciona la plataforma JAVA a través de la JAVA VIRTUAL MACHINE.

En futuras versiones, sería recomendable sustituir PPL por una biblioteca JAVA que tuviera la funcionalidad requerida, para así poder alcanzar el objetivo de plena portabilidad.

4.3.2. JGraphT

JGRAPHT es una biblioteca Java que implementa los tipos de datos y los algoritmos propios de la teoría de grafos. Se trata de un proyecto Open Source que se distribuye bajo la *Lesser General Public License*. La biblioteca puede encontrarse en `http://jgrapht.sourceforge.net`

En nuestro proyecto, las clases de JGraphT han sido una base perfecta para implementar los subprocesos de la evaluación parcial:

- org.jgrapht.DirectedGraph y las clases que lo implementan son idóneas para representar el grafo de dependencias de un SRC.
- org.jgrapht.alg.StrongConnectivityInspector implementa, bajo el patrón *Strategy*, el algoritmo para calcular las componentes fuertemente convexas de un grafo dirigido.

CoveringPointInspector es una clase de nuestro proyecto para descubrir los Covering Points de una CFC. Será entregada como colaboración al proyecto JGraphT, pues puede ser muy útil en otras ramas de la ingeniería. Por ejemplo, serviría para analizar una red de suministro y detectar esos nodos críticos cuyas averías pueden dejar sin luz o agua a una ciudad entera.

[4] `http://gmplib.org`

Recomendamos y animamos a otros programadores a emplear esta biblioteca en proyectos académicos o industriales ya que:

- está programada exclusivamente en Java siguiendo las convenciones de ese lenguaje, sin código o nomenclaturas heredadas, lo que garantiza su portabilidad.
- tiene una amplia funcionalidad, es muy genérica y muy extensible.
- por su facilidad de uso y su detallada documentación.
- y por su capacidad para integrarse en proyectos de mayor envergadura.

Bibliografía

[1] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Proc. of Static Analysis Symposium (SAS)*, LNCS. Springer-Verlag, July 2008.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Cost Equation Systems: a Language–Independent Target Language for Cost Analysis. Technical Report CLIP1/2008.0, Technical University of Madrid, School of Computer Science, UPM, January 2008.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Experiments in Cost Analysis of Java Bytecode. In *BYTECODE'07*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2007. Available online http://www.clip.dia.fi.upm.es/papers/jvm-cost-exp-bytecode07.pdf.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: A Cost and Termination Analyzer for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode)*, Electronic Notes in Theoretical Computer Science, Budapest, Hungary, April 2008. Elsevier.

[5] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis of Java Bytecode. In *International Symposium on Memory Management (ISMM'07)*. ACM Press, 2007.

[6] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006. Available at `http://www.cs.unipr.it/Publications/`. Also published as `arXiv:cs.MS/0612085`, available from `http://arxiv.org/`.

[7] Kent Beck. Test driven development: by example. 2006.

[8] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, LNCS, 2004.

[9] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41, 2004.

[10] G. Román-Díez and G. Puebla. Java bytecode timing cost models. Technical Report CLIP12/2007.0, Technical University of Madrid, School of Computer Science, UPM, December 2007.