Mejora del rendimiento de la depuración declarativa mediante expansión y compresión de bucles*

David Insa[1], **Josep Silva**[2], **César Tomás**[3]

[1] dinsa@dsic.upv.es
[2] jsilva@dsic.upv.es
[3] ctomas@dsic.upv.es
Universitat Politècnica de València
Camino de Vera s/n, E-46022 Valencia, Spain.

Abstract: Uno de los principales objetivos en la depuración es reducir al máximo el tiempo necesario para encontrar los errores. En la depuración declarativa este tiempo depende en gran medida del número de preguntas realizadas al usuario por el depurador y, por tanto, reducir el número de preguntas generadas es un objetivo prioritario. En este trabajo demostramos que transformar los bucles del programa a depurar puede tener una influencia importante sobre el rendimiento del depurador. Concretamente, introducimos dos algoritmos que expanden y comprimen la representación interna utilizada por los depuradores declarativos para representar bucles. El resultado es una serie de transformaciones que pueden realizarse automáticamente antes de que el usuario intervenga en la depuración y que producen una mejora considerable a un coste muy bajo.

Keywords: Depuración declarativa, Árbol de ejecución, *Tree Compression*, *Loop Expansion*

1. Introducción

La depuración es una de las tareas más difíciles y menos automatizadas de la ingeniería del software. Esto se debe al hecho de que los errores están generalmente ocultos tras complejas condiciones que únicamente ocurren en interacciones particulares de componentes software. Normalmente, los programadores no pueden considerar todas las ejecuciones posibles de su software, y precisamente esas ejecuciones no consideradas son las que producen un error. Esta dificultad inherente a la depuración es explicada por Brian Kernighan así:

> *"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"*
>
> The Elements of Programming Style, 2nd edition

* Este trabajo ha sido parcialmente financiado por el *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* con referencia TIN2008-06622-C03-02 y por la *Generalitat Valenciana* con referencia PROMETEO/2011/052. David Insa ha sido parcialmente financiado por el Ministerio de Educación con beca FPU AP2010-4415.

```
    public class Matrix {
      private int numRows;
      private int numColumns;
      private int[][] matrix;

(1)   public Matrix(int numRows, int numColumns) {
        this.numRows = numRows;
        this.numColumns = numColumns;
        this.matrix = new int[numRows][numColumns];
        for (int i = 0; i < numRows; i++)
          for (int j = 0; j < numColumns; j++)
            this.matrix[i][j] = 1;
      }

(2)   public int position(int numRow, int numColumn) {
        return matrix[numRow][numColumn];
      }
    }
```

```
    public class sumMatrix {
(0)   public static void main(String[] args) {
        int result = 0;
        int numRows = 3;
        int numColumns = 3;
        Matrix m = new Matrix(numRows, numColumns);
        for (int i = 0; i < numRows; i++)
          for (int j = 0; j < numColumns; j++)
            result += m.position(i, j);
        System.out.println(result);
      }
    }
```

Figura 1: Programa iterativo escrito en Java

Los problemas causados por los errores del software tienen un coste muy elevado, muchas veces incluso más que el del propio desarrollo del producto. Por ejemplo, en el informe NIST [NIS02] se calcula que los errores de software no detectados antes de la entrega del producto producen un coste de 59 billones de dólares al año a la economía de EEUU.

A pesar de que se han definido muchas técnicas de depuración automáticas, generalmente se han obtenido pobres resultados. Dos excepciones son la Depuración Delta [Art11, YLCZ11] y la Depuración Declarativa [Sha82, Sil11]. En este artículo presentamos una técnica para mejorar el rendimiento de la Depuración Declarativa reduciendo el tiempo de depuración.

La depuración declarativa es una técnica de depuración semi-automática que genera preguntas sobre los resultados obtenidos en diferentes subcomputaciones. De esta forma, el programador responde a las preguntas y, junto con la información del depurador, es capaz de identificar el error en el código fuente de forma precisa. Hablando en términos generales, el depurador descarta aquellas partes del código fuente que están asociadas a computaciones correctas hasta que se aísla una pequeña parte del código (normalmente una función o procedimiento) donde se encuentra el error. Una propiedad interesante de esta técnica es que el programador no necesita ver el código fuente durante la depuración, únicamente necesita saber el resultado obtenido y el esperado por una ejecución con unas determinadas entradas. De esta forma, si tenemos una especificación formal de los componentes software que es capaz de responder a las preguntas, entonces la técnica es totalmente automática.

Explicaremos la técnica mediante un ejemplo. Considérese el programa Java de la Figura 1. Este programa inicializa los elementos de una matriz a 1 y recorre toda la matriz para sumar todos los elementos. El Árbol de Ejecución (AE) es la versión dinámica del grafo de llamadas y representa con un nodo cada ejecución de un método. El AE asociado a este ejemplo se muestra en la Figura 2 (izquierda). Obsérvese que `main` **(0)** llama al constructor de `Matrix` **(1)**, y entonces llama nueve veces a `position` **(2)** dentro de dos bucles iterativos anidados. La depuración declarativa usa una estrategia de navegación para atravesar el AE preguntando al programador sobre la validez de los nodos. Cada nodo contiene una ejecución diferente de un método con sus entradas y salidas. Teniendo en cuenta esta información, el usuario puede marcar el nodo como

Figura 2: AE de los ejemplos de las Figuras 1 (izquierda) y 8 (derecha)

correcto o *incorrecto*[1] Si el nodo está marcado como incorrecto, entonces el error debe estar en el subárbol de este nodo; y debe estar en el resto del árbol si éste está marcado como correcto. Cuando un nodo es marcado como incorrecto, y todos sus hijos están marcados como correctos, entonces este nodo se convierte en un *nodo erróneo* y el depurador presenta el método asociado como causante del error. En el AE de la Figura 2, cada nodo se ha etiquetado (dentro) con un número que se refiere a la función específica que se ha ejecutado (hemos ignorado los argumentos y resultados por no ser de interés para este trabajo). Además, cada nodo tiene asociado un número (fuera) que indica el número de preguntas necesarias para encontrar el error si este nodo es el nodo erróneo. Para calcular este número, hemos considerado la estrategia de navegación *Divide and Query* (D&Q), que siempre selecciona el nodo que mejor divide el AE por la mitad.

Uno de los principales problemas de la depuración declarativa es que puede producir series de preguntas demasiado largas haciendo que la sesión de depuración se extienda demasiado. Además, la depuración declarativa trabaja a nivel de funciones, de forma que el nivel de granularidad del error encontrado es una función. En este trabajo proponemos una nueva técnica que permite a los depuradores declarativos reducir el número de preguntas necesario para detectar un error, además de permitir reducir el nivel de granularidad a bucles, siendo capaz de reportar un bucle de una función como el causante del error. La técnica se basa en diferentes transformaciones del AE de forma que el AE producido puede ser depurado de forma más eficiente usando los algoritmos estándar. Por lo tanto la técnica es conservativa respecto a las implementaciones anteriores y puede ser integrado en cualquier depurador como una fase de preproceso que sólo afecte al AE. La técnica no sólo reduce el número de preguntas realizadas, sino que puede reducir también la complejidad de las mismas. El coste de las transformaciones es muy bajo comparado con el coste de generar todo el AE y, de hecho, son lo suficientemente eficientes como para usarse siempre en todos los depuradores declarativos antes de la fase de exploración del AE ya que, de acuerdo a nuestros experimentos (véase Tabla 1), como término medio reducen el número de

[1] Existen además otros estados posibles (*desconocido*, *de confianza*, etc.) que, por simplicidad, no consideramos en este artículo por no tener ninguna repercusión sobre las técnicas propuestas [Sil11].

preguntas en un 27.65 %.

El resto del artículo se ha organizado de la siguiente manera. En la sección 2 se revisan otras técnicas y trabajos relacionados. La sección 3 presenta algunas definiciones preliminares que se utilizarán en el resto del trabajo. En la sección 4 explicamos nuestra técnica y sus principales aplicaciones, y se introducen los algoritmos utilizados para mejorar la estructura del AE. La corrección de la técnica se demuestra en la sección 5. En la sección 6 presentamos nuestra implementación y algunos experimentos llevados a cabo con programas Java reales. Finalmente, la sección 7 concluye el artículo. Toda la información relacionada con los experimentos, el código fuente de la herramienta, los casos de prueba y otro material puede encontrarse en http://www.dsic.upv.es/~jsilva/DDJ/.

2. Trabajos Relacionados

Reducir el número de preguntas realizadas por los depuradores declarativos es un objetivo prioritario en el campo, y existen bastantes trabajos orientados a alcanzar este objetivo. Muchos de ellos afrontan el problema definiendo diferentes transformaciones del AE que modifican la estructura del mismo haciendo su exploración más eficiente.

Por ejemplo, en [RVMC11], los autores proponen una técnica para mejorar la depuración declarativa de especificaciones Maude cuyo efecto equilibra el AE. Trabajar sobre un AE equilibrado es conveniente cuando la exploración del AE se realiza con estrategias como *Divide and Query* [Sil11], puesto que después de cada respuesta es posible podar cerca de la mitad del árbol, obteniendo así un número de preguntas logarítmico con respecto al número de nodos. La técnica utiliza una transformación que permite al usuario decidir si quiere mantener los pasos de inferencia de transitividad aplicados por el cálculo. Mantener estos pasos equilibra el árbol y, por lo tanto, se realizan menos preguntas en general. Este enfoque está relacionado con nuestra técnica, pero tiene algunos inconvenientes: sólo puede aplicarse cuando tienen lugar las inferencias de transitividad y, por lo tanto, la mayoría de las partes del árbol no se pueden equilibrar, e incluso en estos casos el balanceo sólo afecta a dos nodos. Nuestra técnica, por el contrario, se aplica a los bucles, siendo capaz de equilibrar todo el bucle.

El enfoque más parecido al nuestro es la técnica *Tree Compression* (TC) introducida por Davie y Chitil [DC06]. Se trata también de un enfoque conservativo que transforma el AE en otro equivalente (más pequeño) en el que se pueden encontrar los mismos errores. El objetivo de esta técnica es esencialmente distinto: intenta reducir el tamaño del AE eliminando nodos redundantes, y sólo es aplicable a llamadas recursivas (sólo trata el caso de la recursión directa). Explicaremos esta técnica mediante varios ejemplos.

Ejemplo 1 *Considérese el AE de la Figura 3. Por cada llamada recursiva, TC elimina el nodo hijo asociado a la llamada recursiva y todos los hijos de este nodo pasan a ser hijos del nodo padre. Así, se han eliminado seis nodos de forma que el tamaño del árbol se ha reducido estáticamente. Nótese que el número medio de preguntas (sumatorio del número de preguntas asociadas a cada nodo dividido entre el número total de nodos) ha sido reducido ($\frac{72}{17}$ frente a $\frac{42}{11}$) gracias a la utilización de TC.*

Desafortunadamente, TC no produce siempre buenos resultados. Algunas veces reducir el

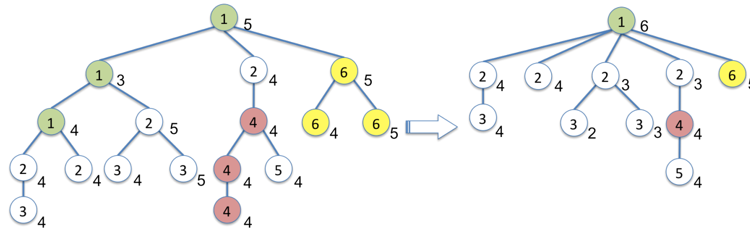

Figura 3: *Tree Compression*

número de nodos causa una estructura del AE que es más difícil de depurar, incrementando así el número de preguntas y produciendo el efecto contrario al deseado.

Ejemplo 2 *Considérese el AE de la Figura 4. En este AE, el número medio de preguntas necesarias para encontrar el error es* $\frac{33}{9}$*. Sin embargo, después de comprimir las llamadas recursivas (nodos oscuros), el número medio de preguntas es incrementado a* $\frac{28}{7}$ *(véase el AE de la izquierda). La razón es que el nuevo AE comprimido no puede podar ningún nodo al tener la estructura completamente plana (cualquier nodo marcado como correcto sólo se podará a sí mismo). Nótese que la estructura original nos permite podar algunos nodos ya que los árboles profundos son más adecuados. Sin embargo, si únicamente comprimimos una de las dos llamadas recursivas, el número de preguntas se reduce a* $\frac{27}{8}$*.*

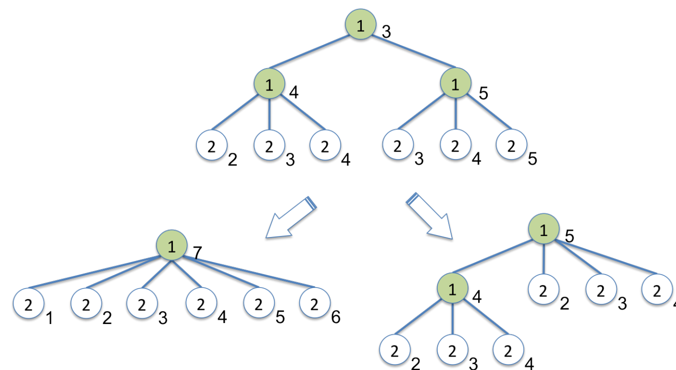

Figura 4: *Tree Compression*

El Ejemplo 2 muestra claramente que TC no debe aplicarse siempre a todas las llamadas recursivas. Sin embargo, las implementaciones existentes siempre comprimen todas ellas [DC06, IS10] puesto que no existe un algoritmo para decidir cuándo aplicar TC y cuándo no. Nuestra técnica soluciona este problema. En particular, una de nuestras transformaciones realiza un análisis para decidir cuándo comprimir llamadas recursivas.

Un enfoque similar a TC es *Declarative Source Debugging* [Cal92] que, en lugar de modificar el árbol, implementa un algoritmo para impedir al depurador seleccionar preguntas relacionadas con los nodos generados por las llamadas recursivas.

Otro enfoque que está relacionado con el nuestro se presentó en [Nil98] donde se introdujo una transformación del código fuente (en lugar del AE) de las listas intensionales de los programas funcionales. Concretamente, esta técnica transforma listas intensionales en un conjunto de funciones equivalentes que implementan la iteración. El AE producido puede ser transformado aún más para eliminar los nodos internos reduciendo el tamaño del AE final como en la técnica TC. Esta técnica es compatible con la nuestra y puede ser aplicada antes de la misma. Es importante destacar que todos los enfoques anteriores fueron definidos en el contexto de los lenguajes funcionales o lógicos. La razón es que la depuración declarativa se definió en el contexto de los lenguajes declarativos y, por lo tanto, gran parte de la investigación se ha llevado a cabo para estos lenguajes. Sin embargo, muchos depuradores declarativos han sido implementados para otros lenguajes como Java (véase [GJ04, CHK06, IS10]), y en estos lenguajes, los bucles se implementan principalmente mediante el uso de iteraciones en lugar de recursión. Por estas razones, muchas de las técnicas anteriores son inútiles para Java, donde las optimizaciones deberían estar orientadas a construcciones iterativas.

Por ejemplo, el AE de la Figura 2 (izquierda) es claramente un AE mal estructurado ya que no permite podar nodos. Fue producido con dos bucles iterativos (anidados) que son una estructura común en los lenguajes imperativos y orientados a objetos. Este tipo de AE es muy común en dichos lenguajes y, por desgracia, no puede ser optimizado con ninguna de las técnicas anteriormente descritas. En las siguientes secciones describimos una técnica capaz de optimizar este tipo de AE.

3. Preliminares

Nuestras transformaciones del AE están basadas en la propia estructura del árbol y en el nombre de la función usada en cada nodo. Esta información es suficiente para identificar todas las cadenas de recursión. Por ello, para el propósito de este trabajo, podemos utilizar una definición de AE en la que los nodos se etiquetan con un número que identifica a una función específica.

Definición 1 (Árbol de Ejecución) Un *Árbol de Ejecución* es un árbol dirigido etiquetado $T = (N,E)$ cuyos nodos N están etiquetados con identificadores de métodos, donde la etiqueta del nodo n es referenciada con $l(n)$. Cada arco $(n \rightarrow n') \in E$ indica que el método asociado a $l(n')$ se invoca durante la ejecución del método asociado a $l(n)$.

Usamos números como identificadores de métodos que identifican unívocamente cada método en el código fuente. Esta simplificación es suficiente para mantener nuestras definiciones y algoritmos precisos, y al mismo tiempo simplificarlos. Dado un AE, la *recursión* o *recursión simple* se representa con una rama de nodos enlazados con la misma etiqueta. La *recursión anidada* ocurre cuando una rama que representa una recursión es descendiente de un nodo en otra recursión. La *recursión múltiple* ocurre cuando un nodo etiquetado con un número n tiene dos o más hijos etiquetados con n.

De aquí en adelante, nos referiremos a las dos estrategias de navegación más usadas en la depuración declarativa: *Top-Down* y D&Q. En ambos casos, siempre nos referiremos implícitamente a la versión más eficiente de ambas técnicas, denominadas (i) *Heaviest First* para Top-Down; la cual siempre recorre el AE desde la raíz hasta las hojas seleccionando siempre el nodo con más

descendientes; y (ii) *Hirunkitti's D&Q* que siempre selecciona el nodo en el AE que divide mejor el AE por la mitad. Un estudio comparativo sobre estas técnicas puede encontrarse en [Sil11].

Para la comparación de estrategias usamos la función $Coste(T,s)$ que computa el número necesario de preguntas (como media) para encontrar el error en un AE T usando la estrategia de navegación s. El peso de un nodo representa el número de nodos contenidos en el subárbol cuya raíz es ese nodo. Nos referimos al peso de un nodo n como w_n.

4. Optimización de AE usando *Tree Compression* y *Loop Expansion*

En esta sección presentamos una nueva técnica para la optimización de los AE que combina dos transformaciones independientes: *Tree Compression* (TC) y *Loop Expansion* (LE). TC fue definida y descrita en [DC06]. Aquí introducimos un algoritmo capaz de decidir cuándo comprimir una rama del AE en cualquier caso, incluyendo recursión simple, recursión anidada y recursión múltiple. También discutimos cómo las estrategias se ven afectadas por TC. La otra técnica introducida es LE que, esencialmente, transforma un bucle iterativo en una función recursiva equivalente; y a continuación, agrupa adecuadamente las llamadas recursivas para obtener un nuevo AE tan mejorado como sea posible. Explicaremos cada técnica por separado y mostraremos algoritmos para ellas que pueden trabajar de manera independiente.

4.1. Cuándo aplicar *Tree Compression*

Tree Compression fue propuesta como una técnica general para la depuración declarativa. Sin embargo, fue definida en el contexto de un lenguaje funcional (Haskell) y con el uso de una estrategia particular (Hat Delta). Los propios autores afirmaron que TC puede producir árboles anchos que son difíciles de depurar y, por esta razón, definieron heurísticas que evitaban preguntar sobre la misma función de forma repetida. Desafortunadamente, estas heurísticas sólo son útiles en presencia de llamadas recursivas.

Así, a pesar de que TC puede ser muy útil, también puede producir malos AE como se mostró en la Figura 4. Desafortunadamente, los autores de TC no estudiaron cómo trabaja esta técnica con otras estrategias (más extendidas) como Top-Down o D&Q, de forma que no está claro cuándo usar TC con estas estrategias. Para estudiar esto, consideramos el caso más general de la recursión en un AE como se muestra en la Figura 5, donde las nubes representan conjuntos posiblemente vacíos de subárboles, los nodos oscuros son la cadena de recursión con una longitud de n llamadas, $n \geq 2$, y las etiquetas de los nodos son identificadores.

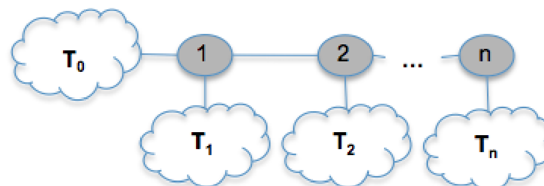

Figura 5: Cadena de recursión en un AE

Debe quedar claro que la cadena de recursión puede ser útil para podar nodos del árbol. Por ejemplo, en la Figura 5, si preguntamos por el nodo n/2, se pueden podar $n/2$ subárboles. En el caso de que los subárboles $T_i, 1 \leq i \leq n$, estén vacíos, entonces no es posible la poda y consecuentemente TC debe ser utilizado. Por lo tanto, podemos concluir que *cada llamada recursiva sin hijos, o cuyo único hijo es otra llamada recursiva, debe ser comprimida*. Este resultado puede ser formalmente establecido para Top-Down de la siguiente manera:

Teorema 1 *Sea T un AE con una cadena de recursión $R = n_1 \to n_2 \to \ldots \to n_m$ donde el único hijo de un nodo n_i, $1 \leq i \leq m-1$, es n_{i+1}. Y sea T' un AE equivalente a T excepto que el nodo n_i ha sido comprimido. Entonces, $Coste(T', Top\text{-}Down) < Coste(T, Top\text{-}Down)$.*

Este teorema ofrece una oportunidad para mejorar de forma estática la estructura de un AE utilizando TC. Sin embargo, TC no es la panacea, y es necesario identificar en qué casos se debe utilizar. D&Q es un buen ejemplo de estrategia en el que TC tiene un efecto negativo.

Tree Compression para D&Q

En general, cuando se depura un AE con la estrategia D&Q, TC no debe ser aplicado, salvo en el caso mostrado por el Teorema 1 ($T_i, 1 \leq i \leq n$, están vacíos). La razón es que D&Q puede saltar a cualquier nodo del AE sin seguir un camino predefinido. Esto permite a D&Q preguntar acerca de cualquier nodo de la cadena de recursividad, sin necesidad de preguntar acerca de los nodos anteriores de la cadena. Nótese que esto no sucede en otras estrategias como Top-Down. Por lo tanto, D&Q tiene la capacidad de utilizar la cadena de recursión como un medio para dividir las iteraciones por la mitad podando nodos.

Dado el AE de la Figura 5, el único caso en el que D&Q no puede utilizar la cadena de recursividad para podar nodos es cuando los conjuntos de subárboles de $T_1 \ldots T_n$ están vacíos. En ese caso, los únicos nodos que pueden podarse son los de la propia cadena y, por consiguiente, es mejor podarlos estáticamente mediante TC. Sin embargo, si los subárboles contienen al menos un nodo, entonces TC *no* debe ser utilizado puesto que la estructura del árbol empeoraría tal y como se muestra en la Figura 6.

Obsérvese en la figura que, a excepción de las cadenas de recursión muy pequeñas (por ejemplo, $n \leq 3$), D&Q puede obtener ventaja de la cadena de recursividad para podar la mitad de las iteraciones. Cuanto mayor es n más nodos se podan. Téngase en cuenta que incluso con un solo hijo en los nodos de la cadena de recursión (por ejemplo, $T_i, 1 \leq i \leq n$ es un único nodo), D&Q puede podar nodos. Por lo tanto, si añadimos más nodos a los subárboles Ti, entonces más nodos pueden ser podados y D&Q se comporta aún mejor. De esta forma podemos concluir que TC no debe ser utilizado en combinación con D&Q excepto en el caso caracterizado en el Teorema 1.

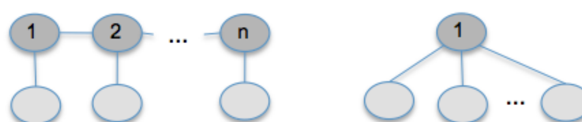

Figura 6: TC aplicado a una secuencia de llamadas recursivas

Tree Compression para Top-Down

En el caso de estrategias basadas en Top-Down, no es trivial en absoluto decidir cuándo aplicar TC y cuándo no aplicarlo. Considerando de nuevo el AE de la Figura 5, hay tres factores que deben tenerse en cuenta para decidir cuándo aplicar TC:

1. la longitud n de la cadena recursiva,
2. el tamaño de cada uno de los árboles $T_i, 1 \leq i \leq n$, y
3. la estrategia de navegación usada.

Con el fin de decidir en qué casos se debe aplicar TC, proporcionamos el Algoritmo 1 que recibe un AE y determina qué nodos de las cadenas de recursión deben ser comprimidos.

En esencia, el Algoritmo 1 analiza para cada posible cadena de recursión cual es el efecto de aplicar TC, y en caso de resultar en una mejora, lo aplica. Esto no se hace con toda la cadena de recursión a la vez, si no que se hace para cada par de nodos recursivos. De tal forma que el resultado final puede ser comprimir solamente una parte (o varias) de la cadena de recursión.

Para ello, la variable *recs* contiene inicialmente todos los nodos del AE que tienen un hijo recursivo. Cada uno de estos nodos se procesa con el bucle de la línea 1 en un proceso desde las hojas hasta la raíz (línea 2). Es decir, primero se procesan aquellos nodos más cercanos a las hojas. Para tratar la recursión múltiple, se usan los bucles de las líneas 5 y 8, que procesan cada rama recursiva de manera independiente y calculan qué ramas deben ser comprimidas para poder obtener una mejora. Dicha mejora se almacena en la variable *improvement*.

El algoritmo hace uso de dos funciones cuyo código ha sido incluido (Cost y Compress), y tres funciones cuyo código no ha sido incluido por ser trivial: la función Children obtiene el conjunto de hijos de un nodo en el AE (i.e., $\text{Children}(m) = \{n \mid (m \rightarrow n) \in E\}$); la función Sort recibe un conjunto de nodos y obtiene una secuencia ordenada donde los nodos han sido ordenados de forma decreciente por sus pesos; y la función Pos recibe un nodo y una secuencia de nodos y devuelve la posición de ese nodo en la secuencia.

Dados dos nodos *parent* y *child* candidatos a ser comprimidos con TC, el algoritmo primero ordena los hijos tanto de *parent* como de *child* (líneas 9-10) en el orden en el que Top-Down preguntaría por ellos (ordenados por su peso). A continuación, se combinan los hijos de ambos nodos para simular una TC (línea 11). Finalmente, se compara el número de preguntas realizadas en ambas opciones (línea 12).

La ecuación de la línea 12 es una de las principales contribuciones de este algoritmo puesto que determina cuándo aplicar TC entre dos nodos de una cadena con la estrategia Top-Down. Esta ecuación depende de la fórmula (línea 21 en la función Cost) usada para calcular el coste medio al aplicar Top-Down.

4.2. Loop Expansion

En general, las llamadas recursivas dividen el AE en diferentes iteraciones representadas con subárboles cuyas raíces pertenecen a la cadena de recursión. Esta estructura es muy conveniente ya que permite la poda de diferentes iteraciones gracias a la cadena de recursión. Por lo tanto, la recursividad es beneficiosa para la depuración declarativa, excepto en los casos expuestos en

Algorithm 1 Tree Compression Optimizado

Entrada: An ET $T = (N, E)$
Salida: An ET T'
Inicialización: $T' = T$ and $recs = \{n \mid n, n' \in N \wedge (n \rightarrow n') \in E \wedge l(n) = l(n')\}$

begin
1) **while** $(recs \neq \emptyset)$
2) **take** $n \in recs$ such that $\nexists n' \in recs$ with $(n \rightarrow n') \in E^+$
3) $recs = recs \setminus \{n\}$
4) $parent = n$
5) **do**
6) $maxImprovement = 0$
7) $children = \{c \mid (n \rightarrow c) \in E \wedge l(n) = l(c)\}$
8) **for each** $child \in children$
9) $pchildren = \text{Sort}(\text{Children}(parent))$
10) $cchildren = \text{Sort}(\text{Children}(child))$
11) $comb = \text{Sort}((pchildren \cup cchildren) \setminus \{child\})$
12) $improvement = \frac{Cost(pchildren)+Cost(cchildren)}{w_{parent}} - \frac{Cost(comb)}{w_{parent}-1}$
13) **if** $(improvement > maxImprovement)$
14) $maxImprovement = improvement$
15) $bestNode = child$
16) **end for each**
17) **if** $(maxImprovement \neq 0)$
18) $T' = \text{Compress}(T', parent, bestNode)$
19) **while** $(maxImprovement \neq 0)$
20) **end while**
end
return T'

function Cost$(sequence)$
begin
21) **return** $\sum\{\text{Pos}(node, sequence) * w_{node} \mid node \in sequence\} + |sequence|$
end

function Compress$(T = (N, E), parent, child)$
begin
22) $nodes = \text{Children}(child)$
23) $E' = E \setminus \{(child \rightarrow n) \in E \mid n \in nodes\}$
24) $E' = E' \cup \{(parent \rightarrow n) \mid n \in nodes\}$
25) $N' = N \setminus \{child\}$
end
return $T' = (N', E')$

la sección anterior. Por el contrario, los bucles iterativos producen árboles muy anchos, donde todas las iteraciones son representadas como árboles hijos de una misma raíz. En esta estructura, no es posible podar más de una iteración a la vez, siendo la depuración de estos árboles muy costosa.

Para solucionar este problema, en esta sección se presenta una nueva técnica que transforma los bucles iterativos en funciones recursivas equivalentes. Debido a que la iteración es más eficiente que la recursión, existen muchos enfoques para transformar funciones recursivas en bucles

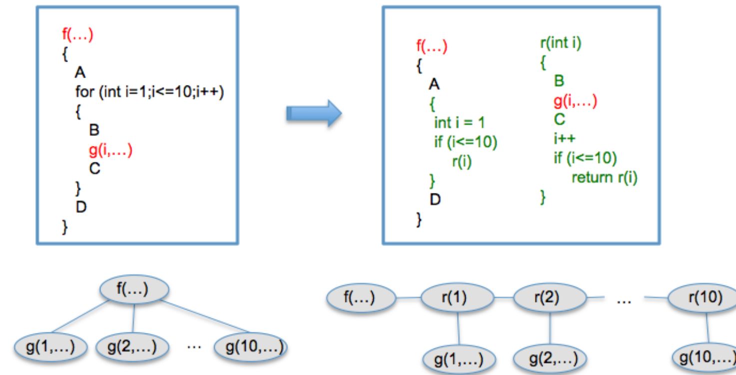

Figura 7: Transformación de un bucle iterativo en una función recursiva equivalente

iterativos (por ejemplo, [HK92, LS00]). Sin embargo, existen muy pocos métodos para transformar bucles iterativos en funciones recursivas equivalentes. Una excepción es la introducida en [YAK00] para mejorar el rendimiento en las jerarquías de memoria multi-nivel. No obstante, no tenemos conocimiento de ningún algoritmo de este tipo propuesto para Java o para cualquier otro lenguaje orientado a objetos. Así, hemos implementado este algoritmo y lo hemos puesto a disposición pública para la comunidad [IS12]. Este algoritmo es la base de LE y básicamente transforma cada bucle iterativo en una función recursiva equivalente. La transformación es ligeramente distinta para cada tipo de bucle (`for`, `foreach`, `while` o `do`).

En el caso de los bucles `for`, se puede explicar con el código de la Figura 7 donde A, B, C y D representan bloques de código. Si observamos el AE transformado vemos que cada iteración se representa con un nodo de la cadena de recursión diferente $r(1) \rightarrow r(2) \rightarrow \ldots \rightarrow r(10)$, de forma que es posible detectar un error en una iteración aislada o en el bucle entero. Esto significa que, en el caso de que la función f tuviera un error, gracias a la transformación, el depurador podría detectar el error en el código B + C o A + D. Nótese que esto no es posible en el AE original, donde el depurador siempre reportaría que A + B + C + D tiene un error. Éste es un resultado interesante puesto que permite aumentar el nivel de granularidad de los errores encontrados, siendo el depurador capaz de detectar los errores dentro de bucles, y no sólo dentro de métodos.

La recursión anidada aumenta las posibilidades de poda. Por ejemplo, la clase Matrix de la Figura 1 puede ser automáticamente transformada[2] al código de la Figura 8. El AE asociado al programa transformado se muestra en la Figura 2 (derecha). Es fácil darse cuenta de que hay una cadena de recursión para cada bucle ejecutado y, por tanto, ahora es posible eliminar bucles, iteraciones, o llamadas individuales dentro de una iteración.

Una vez que LE ha expandido todos los bucles, utilizamos el Algoritmo 1 para volver a comprimir aquellas (sub)cadenas de recursión generadas que no debieron ser expandidas. De esta manera, se consigue tener una representación ideal para cada uno de los bucles del programa.

[2] Por claridad, en la figura hemos cambiado los nombres de las funciones recursivas generadas por `sumRows` y `sumColumns`. En la transformación, si el bucle tiene una etiqueta de bucle (una *label*), se usa esa etiqueta como nombre del bucle. Si la etiqueta no existe, entonces cada bucle tiene como nombre el nombre del método asociado seguido de "_bucleN", donde N es un autonúmerico. Durante la depuración, el depurador muestra al usuario el código del bucle y le permite asociarle otro nombre si éste lo desea.

```
    public class sumMatrix {
(0)   public static void main() {
         int result = 0;
         int numRows = 3;
         int numColumns = 3;
         Matrix m = new Matrix(numRows, numColumns);
         // For loop
         { // Init for loop
            int i = 0;
            // First iteration
            if (i < numRows) {
               Object[] res = sumMatrix.sumRows(m, i, numRows, numColumns, result);
               result = (Integer)res[0];
            }
         }
         System.out.println(result);
      }
(3)   private static Object[] sumRows(Matrix m, int i, int numRows, int numColumns, int result) {
         // For loop
         { // Init for loop
            int j = 0;
            // First iteration
            if (j < numColumns) {
               Object[] res = sumMatrix.sumColumns(m, i, j, numColumns, result);
               result = (Integer)res[0];
            }
         }
         // Update for loop
         i++;
         // Next iteration
         if (i < numRows)
            return sumMatrix.sumRows(m, i, numRows, numColumns, result);
         return new Object[]{result};
      }
(4)   private static Object[] sumColumns(Matrix m, int i, int j, int numColumns, int result) {
         result += m.position(i, j);
         // Update for loop
         j++;
         // Next iteration
         if (j < numColumns)
            return sumMatrix.sumColumns(m, i, j, numColumns, result);
         return new Object[]{result};
      }
    }
```

Figura 8: Versión recursiva del código de la Figura 1

5. Corrección

En esta sección se demuestra que después de nuestras transformaciones, todos los errores que puedan detectarse en el AE original todavía se pueden detectar en el AE transformado. Un resultado aún más interesante es que el AE transformado puede contener más nodos con errores que el AE original. En cuanto a TC, su corrección se ha demostrado en [DC06]. Nuestro algoritmo no influye en esta propiedad de corrección, ya que sólo decide qué nodos deben ser comprimidos y el algoritmo de TC utilizado es el estándar. La completitud y la corrección de LE se formulan a continuación. Las demostraciones asociadas pueden encontrarse en el Apéndice 1.

Teorema 2 *[Completitud] Sea P un programa y T su AE asociado, sea P' el programa obtenido al aplicar* Loop Expansion *a P, y sea T' el AE asociado a P'. Para cada nodo erróneo en T, hay al menos un nodo erroneo en T'.*

Teorema 3 *[Corrección] Sea P un programa y T su AE asociado, sea P' el programa obtenido al aplicar* Loop Expansion *a P, y sea T' el AE asociado a P'. Si T' contiene un nodo erróneo asociado al código $f \subseteq P$, entonces T contiene un nodo erróneo asociado al código $g \subseteq P$ y $f \subseteq g$.*

Programa	Nodos				LE	Tiempo		Preguntas				%	
	AE	LE	TC_{ori}	TC_{opt}		LE	TC	AE	LE	TC_{ori}	TC_{opt}	LETC	TC
Factoricer	55	331	51	51	5	151	105	11.62	8.50	7.35	7.35	63.25	100.0
Classifier	25	57	22	24	3	184	4	8.64	6.19	6.46	6.29	72.80	97.36
LegendGame	87	243	87	87	10	259	31	12.81	8.28	11.84	11.84	92.43	100.0
Romanic	121	171	112	113	3	191	12	16.24	7.74	10.75	9.42	58.00	87.62
FibRecursive	5378	6192	98	101	12	251	953	15.64	12.91	9.21	8.00	51.15	86.86
FactTrans	197	212	24	26	3	181	26	10.75	7.88	6.42	5.08	47.26	79.13
BinaryArrays	141	203	100	100	5	172	79	12.17	7.76	7.89	7.89	64.83	100.0
FibFactAna	178	261	44	49	7	202	33	7.90	8.29	8.50	6.06	76.71	71.29
RegresionTest	13	121	15	15	5	237	4	4.77	7.17	4.20	4.20	88.05	100.0
BoubleFibArrays	16	164	10	10	10	213	27	9.31	8.79	4.90	4.90	52.63	100.0
StatsMeanFib	19	50	23	23	6	195	21	7.79	8.12	6.78	6.48	83.18	95.58
Integral	5	8	8	8	3	152	2	6.80	5.75	7.88	5.88	86.47	74.62
TestMath	3	5	3	3	3	195	2	7.67	6.00	9.00	7.67	100.0	85.22
TestMath2	92	2493	13	13	3	211	607	14.70	11.54	15.77	12.77	86.87	80.98
Figures	2	10	10	10	24	597	13	9.00	7.20	6.60	6.60	73.33	100.0
FactCalc	128	179	75	75	3	206	46	8.45	7.60	7.96	7.96	94.20	100.0
SpaceLimits	95	133	98	100	15	786	10	36.26	12.29	18.46	14.04	38.72	76.06

Tabla 1: Resumen de los experimentos realizados

De los Teoremas 2 y 3 obtenemos un corolario interesante que nos indica que el árbol transformado puede encontrar más errores que el AE original.

Corolario 1 *Sea P un programa y T su AE asociado, sea P' el programa obtenido al aplicar* Loop Expansion *a P, y sea T' el AE asociado a P'. Si T contiene n nodos erróneos, entonces T' contiene n' nodos erróneos con $n \leq n'$.*

6. Implementación

Hemos implementado el algoritmo TC original y el optimizado para transformar árboles de ejecución, y el algoritmo LE de transformación de Java. Todos ellos han sido integrados en el Depurador Declarativo para Java DDJ [IS10, GRS06]. El código fuente de las transformaciones es accesible públicamente en la dirección: http://www.dsic.upv.es/~jsilva/DDJ/.

La implementación de ambos algoritmos se ha hecho en Java. TC tiene aproximadamente 90 líneas de código, y LE alrededor de 1700. Una vez implementadas, hemos realizado varias series de experimentos para medir el impacto que ambas técnicas tienen sobre el rendimiento del depurador. Los resultados obtenidos se resumen en la Tabla 1.

La primera columna de la Tabla 1 indica el nombre del programa depurado. Cada uno de los programas ha sido evaluado asumiendo que el error podría estar en cualquiera de los nodos del árbol de ejecución. Esto significa que cada fila de la tabla es la media de un conjunto de experimentos. Por ejemplo, para evaluar el coste de depurar `Factoricer`, éste se ha depurado 55+331+51+51=488 veces, asumiendo que el error estaba en cada nodo y se ha obtenido la media. Para cada programa, la columna `nodos` muestra el número de nodos que tiene el AE original (`AE`), el AE tras aplicar LE (`LE`), el AE tras aplicar LE primero y después TC en su versión original comprimiendo todos los nodos posibles (TC_{ori}), y el AE tras aplicar LE primero y después TC en su versión optimizada usando el Algoritmo 1 (TC_{opt}); la columna `LE` muestra el número total de bucles expandidos para cada programa; la columna `Tiempo` muestra el tiempo necesario medido en milisegundos para aplicar las transformaciones; la columna `Preguntas` muestra el número medio de preguntas realizadas en cada programa y para cada AE; finalmente, la co-

lumna `(%)` muestra por un lado el porcentaje de preguntas que se realiza tras aplicar nuestras transformaciones (LE y TC) con respecto al AE original (`LETC`); y por otro lado el porcentaje de preguntas que se realiza al usar el Algoritmo 1 para decidir cuándo aplicar TC, con respecto a aplicar siempre TC (`TC`). Como conclusión se observa que utilizar nuestras transformaciones produce una reducción del 27.65 % en el número de preguntas realizadas por el depurador. Por otro lado, el uso del Algoritmo 1 para decidir cuándo aplicar TC y cuando no, también tiene un impacto importante en la reducción del número de preguntas con una media de 9.72 %.

7. Conclusiones

Dos de los principales problemas de la depuración declarativa son (1) que puede generar demasiadas preguntas para encontrar un error, y (2) que una vez que el error se ha encontrado, el depurador reporta un método completo como el código erróneo. En este trabajo hemos dado un paso adelante para resolver estos problemas. Por un lado, hemos introducido técnicas que reducen el número de preguntas mejorando la estructura del AE. Esto se consigue con dos transformaciones llamadas *Tree Compression* y *Loop Expansion*. Por otra parte, *Loop Expansion* también aborda el segundo problema, y nos permite la detección de errores en bucles (y no sólo en métodos) aumentando así el nivel de granularidad del error.

Acknowledgements: Queremos agradecer a Rafael Caballero y Adrián Riesco los comentarios y discusiones productivas realizados en las fases iniciales de este trabajo.

Referencias

[Art11] C. Artho. Iterative Delta Debugging. *International Journal on Software Tools for Technology Transfer (STTT)* 13(3):223–246, 2011.

[Cal92] M. Calejo. *A Framework for Declarative Prolog Debugging*. PhD thesis, New University of Lisbon, 1992.

[CHK06] R. Caballero, C. Hermanns, H. Kuchen. Algorithmic Debugging of Java Programs. In *Proc. of the 2006 Workshop on Functional Logic Programming (WFLP'06)*. Pp. 63–76. Electronic Notes in Theoretical Computer Science, 2006.

[DC06] T. Davie, O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*. Pp. 27–40. April 2006.

[GJ04] P. Gestwicki, B. Jayaraman. JIVE: Java Interactive Visualization Environment. In *OOPSLA'04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. Pp. 226–228. ACM Press, New York, NY, USA, 2004.

[GRS06] F. González-Blanch, F. Roses, S. Serrano. Depurador Declarativo de programas JAVA. In *Master's thesis Universidad Complutense de Madrid*. Available from: http://eprints.ucm.es/9114/, 2006.

[HK92] P. G. Harrison, H. Khoshnevisan. A new approach to recursion removal. *Theor. Comput. Sci.* 93(1):91–113, Feb. 1992.
doi:10.1016/0304-3975(92)90213-Y

[IS10] D. Insa, J. Silva. An Algorithmic Debugger for Java. *In Proc. of the 26th IEEE International Conference on Software Maintenance* 0:1–6, 2010.

[IS12] D. Insa, J. Silva. A Transformation of Iterative Loops into Recursive Loops. Technical report DSIC/05/12, Universitat Politécnica de Valencia, 2012. Available from URL: http://www.dsic.upv.es/~jsilva/research.htm#techs.

[IST12] D. Insa, J. Silva, C. Tomás. Mejora del rendimiento de la depuración declarativa mediante expansión y compresión de bucle. Technical report DSIC/07/12, Universitat Politécnica de Valencia, 2012. Available from URL: http://www.dsic.upv.es/~jsilva/research.htm#techs.

[LS00] Y. A. Liu, S. D. Stoller. From recursion to iteration: what are the optimizations? In *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*. PEPM'00, pp. 73–82. ACM, New York, NY, USA, 2000.
doi:10.1145/328690.328700

[Nil98] H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.

[NIS02] NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing. In Standards and Technology (eds.), *NIST Planning Report 02-3*. May 2002.

[RVMC11] A. Riesco, A. Verdejo, N. Martí-Oliet, R. Caballero. Declarative Debugging of Rewriting Logic Specifications. *Journal of Logic and Algebraic Programming*, September 2011.
doi:http://dx.doi.org/10.1016/j.jlap.2011.06.004

[Sha82] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.

[Sil11] J. Silva. A Survey on Algorithmic Debugging Strategies. *Advances in Engineering Software* 42(11):976–991, Nov. 2011.
doi:10.1016/j.advengsoft.2011.05.024

[YAK00] Q. Yi, V. Adve, K. Kennedy. Transforming Loops to Recursion for Multi-Level Memory Hierarchies. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*. Pp. 169–181. 2000.

[YLCZ11] K. Yu, M. Lin, J. Chen, X. Zhang. Towards Automated Debugging in Software Evolution: Evaluating Delta Debugging on Real Regression Bugs from Developers' Perspectives. *Journal of Systems and Software(JSS)* 85, October 2011.
http://dx.doi.org/10.1016/j.jss.2011.10.016

Apéndice 1: Demostraciones de los resultados teóricos

En esta sección se presentan las demostraciones de los Teoremas 1, 2 y 3 y del Corolario 1.

Teorema 1 *Sea T un AE con una cadena de recursión $R = n_1 \to n_2 \to \ldots \to n_m$ donde el único hijo de un nodo n_i, $1 \leq i \leq m-1$, es n_{i+1}. Y sea T' un AE equivalente a T excepto que el nodo n_i ha sido comprimido. Entonces, $Coste(T'$,Top-Down$) < Coste(T$,Top-Down$)$.*

Demostración. Consideremos los dos nodos que forman la subcadena a comprimir. Para la demostración podemos llamarlos $n_1 \to n_2$. Primeramente, el número de preguntas necesarias para encontrar el error en algún antecesor de n_1 es el mismo tanto si comprimimos como si no la subcadena. Entonces, basta con demostrar que $Coste(T_{1c},\text{Top-Down}) < Coste(T_1,\text{Top-Down})$ donde T_1 es el subárbol cuya raíz es n_1 y T_{1c} es el subárbol cuya raíz es n_1 después de comprimir el árbol.
Asumimos que n_2 tiene j hijos. Así, llamamos T_2 al subárbol cuya raíz es n_2, y T_{2_i} al subárbol cuya raíz es el i-ésimo hijo de n_2. Entonces,

$$Coste(T_2,\text{Top-Down}) = \frac{(j+1) + \sum_{i=1}^{j} |T_{2_i}| * (i + Coste(T_{2_i},\text{Top-Down}))}{|T_2|}$$

En este punto, se necesitan $(j+1)$ preguntas para encontrar el error en n_2. Para alcanzar al hijo de n_2, el propio n_2 y los anteriores $i-1$ hijos deben ser contestados primero, por lo que necesitamos añadir i a $Coste(T_{2_i},\text{Top-Down})$. Finalmente, $|T_x|$ representa el número de nodos en el (sub)árbol T_x.
De esta forma, $Coste(T_{1c},\text{Top-Down}) = Coste(T_2,\text{Top-Down})$
y $Coste(T_1,\text{Top-Down}) = \frac{2 + |T_2| * (i + Coste(T_2,\text{Top-Down}))}{|T_2|+1}$
Claramente, $Coste(T_{1c},\text{Top-Down}) < Coste(T_1,\text{Top-Down})$ y, por lo tanto, el teorema se cumple. □

Teorema 2 *Sea P un programa y T su AE asociado, sea P' el programa obtenido al aplicar* Loop Expansion *a P, y sea T' el AE asociado a P'. Para cada nodo erróneo en T, hay al menos un nodo erroneo en T'.*

Demostración. Probaremos el teorema para un nodo arbitrario n en T asociado con una función f. En primer lugar, como LE sólo transforma bucles iterativos en funciones recursivas, todas la funciones ejecutadas en T se ejecutan también en T'. Esto significa que cada nodo en T tiene un nodo equivalente en T' que representa la misma (sub)computación. Por tanto, podemos llamar n' al nodo que representa en T' la misma ejecución que n en T. Como n es erróneo, entonces n es incorrecto y todos los hijos de n (si existen) son correctos. Por otra parte, si f no contiene un bucle, entonces LE no tiene efecto en el código de f, de forma que n y n' tienen exactamente los mismos hijos, con lo que, trivialmente, n' es también erróneo en T'. Si asumimos la existencia de un bucle en f, entonces tendremos una situación como la mostrada en la Figura 7. Podemos considerar para la prueba que el AE de la izquierda es el subárbol de n y el AE de la derecha es el subárbol de n'. Entonces, como n es erróneo, todos los nodos etiquetados con g son correctos (en ambos AE) y, por consiguiente, o bien n', o los nodos etiquetados con r, son erróneos. □

Teorema 3 *Sea P un programa y T su AE asociado, sea P' el programa obtenido al aplicar* Loop Expansion *a P, y sea T' el AE asociado a P'. Si T' contiene un nodo erróneo asociado al código $f \subseteq P$, entonces T contiene un nodo erróneo asociado al código $g \subseteq P$ y $f \subseteq g$.*

Demostración. De acuerdo a la demostración del Teorema 2, cada nodo en T tiene un nodo asociado en T'. Sea n el nodo erróneo en T y sea n' el nodo erróneo asociado en T'. Si f no tiene un bucle, entonces tanto n como n' apuntan a la misma función (f) y, por lo tanto, el teorema se cumple de manera trivial. Si f contiene un bucle que ha sido expandido, entonces, tal y como se expone en la demostración del Teorema 2, n' o uno de sus descendientes (n'') que representan iteraciones del bucle son erróneos. Sabemos que el código de n'' es el código del bucle que esta incluido en el código de f. Por lo tanto, en todos los casos $f \subseteq g$. □

Corolario 1 *Sea P un programa y T su AE asociado, sea P' el programa obtenido al aplicar* Loop Expansion *a P, y sea T' el AE asociado a P'. Si T contiene n nodos erróneos, entonces T' contiene n' nodos erróneos con $n \leq n'$.*

Demostración. La demostración es trivial a partir de los Teoremas 2 y 3. Por una parte, la igualdad se asegura con el Teorema 2 porque por cada nodo erróneo en T hay al menos un nodo erróneo en T'. Por otra parte, si un nodo en T está asociado a una función cuyo código contiene más de un bucle que ha sido expandido, entonces T' puede contener más de un nodo erróneo nuevo que no está presente en T. □