Cálculo automático de la eficiencia asintótica de programas imperativos

Xavier Franch
franch@lsi.upc.es
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
c/Jordi Girona Salgado, 1-3 (Campus Nord)
08034 Barcelona

Joan Vancells
vancells@gregal.euv.upc.es
Estudis Universitaris de Vic
Escola Universitària Politècnica de Osona
c/ de Miramarges, s/n
08500 Vic (Barcelona)

Resumen

Se presenta en este artículo una propuesta para el cálculo automático del tiempo de ejecución de los programas imperativos, medido con la notación asintótica O grande, y considerando el peor de los casos en el comportamiento de los programas. Para ello, se propone tratar el tiempo de ejecución como un atributo sintetizado de la gramática, y damos las reglas de cálculo adecuadas para un lenguaje de trabajo representativo del paradigma imperativo, destacando la eventual necesidad de definir reglas parametrizadas y de añadir en los programas información relativa a la eficiencia. Se introduce asimismo el concepto de esqueleto para especificar patrones de combinación de operaciones con una eficiencia potencialmente diferente de la que se calcularía con las reglas estándar. Los resultados de la eficiencia se ofrecen simplificados al máximo gracias a la aplicación de unas propiedades de los operadores de la notación O.

1. Introducción

El análisis de la eficiencia de los programas es un campo cuya existencia se remonta al final de la década de los 60. En los años siguientes comienzan a formularse algunas propuestas para la automatización de dicho análisis, hasta que [Web75] presenta el primer sistema de amplia difusión para el cálculo totalmente automático de la eficiencia de programas Lisp, aunque con importantes restricciones en el tipo de algoritmos a analizar y en la clase de manipulaciones simbólicas posibles. El resultado del algoritmo consistía en determinar el tiempo necesario para ejecutarlo en función del número de operaciones elementales necesarias; este tipo de análisis de la eficiencia se bautizó como microanálisis. Dentro de este ámbito, destacamos dos propuestas más. En [HC88] se estudia el cálculo de eficiencia de programas funcionales, aunque más en la línea de verificación de aserciones de rendimiento que de cálculo propiamente dicho. En [FSZ89, Zim89, FSZ91] se describe un sistema para el análisis automático de clases bien definidas de algoritmos que operan sobre estructuras de datos descomponibles, cuyo funcionamiento se entronca con metodologías de análisis combinatorio basado en las correspondencias entre la estructura de los tipos de datos y las funciones generatrices.

Como alternativa al microanálisis, nos encontramos con la técnica de macroanálisis, que expresa el tiempo de ejecución mediante notaciones asintóticas y no mediante valores exactos. Además, y debido a la complejidad de los resultados obtenidos cuando se considera el caso promedio, las técnicas automatizadas de macroanálisis acostumbran a considerar el peor de los casos en el comportamiento de los algoritmos. Si bien este enfoque proporciona resultados menos precisos, las herramientas a construir son capaces de tratar programas más complejos y, así, su aplicación en la ingeniería del *software* a gran escala es factible. Destacamos aquí los trabajos [LeM88, Ros89]. El primero de ellos presenta un sistema para la generación automática de funciones de complejidad para un subconjunto del lenguaje FP, que se basa en técnicas de transformación de programas, y que usa una librería (ampliable) con más de mil reglas de transformación (precisamente, el punto crítico consiste en dar con la regla adecuada en cada momento). La segunda propuesta presenta un sistema para derivar cotas superiores de la eficiencia para programas escritos en un subconjunto de Lisp.

En resumen, podríamos decir que los trabajos que conocemos sobre cálculo automático de la eficiencia de los programas se centran en lenguajes funcionales, debido a la facilidad de conectar su análisis con conceptos matemáticos conocidos y al estudio en profundidad que se ha realizado de la mayoría de problemas derivados (por ejemplo, en la resolución de ecuaciones de recurrencia). En este artículo, presentamos una propuesta que se centra en el macroanálisis de programas imperativos de complejidad no trivial, con el propósito de obtener una herramienta aplicable en el marco de la programación con componentes [Fra97+]. Mediremos la eficiencia usando una notación asintótica llamada O grande y considerando también el peor de los casos.

El resto del artículo se organiza como sigue. En la sección 2, se presenta la notación O grande y se introducen diversas reglas de simplificación. La sección 3 define el lenguaje de trabajo que usaremos a lo largo del artículo. En las secciones 4 y 5 introducimos las construcciones que nos permiten establecer la eficiencia de los programas, mediante un atributo sintetizado de la gramática y mediante lo que denominamos esqueletos, que nos permiten identificar patrones particulares de comportamiento. Finalmente, la sección 6 expone las conclusiones y el trabajo futuro.

2. La notación asintótica O grande

2.1 Definición

En esta sección, vamos a definir con precisión la notación que utilizamos para medir la complejidad de los programas (en cuanto a su tiempo de ejecución), la conocida como O grande (*big-Oh*) o, simplemente, O. La notación O, proveniente de la matemática clásica, fue propuesta en [Knu76] en el ámbito de la programación, y posteriormente transformada en [Bra85], que la define como:

$$O(f) = \{g: \mathcal{N}^+ \to \mathcal{N}^+ / \exists c_0, n_0 \in \mathcal{N}^+: \forall n \geq n_0: g(n) \leq c_0 f(n)\},$$

siendo $\mathcal{N}^+$ el dominio de los números naturales diferentes del cero, y siendo f: $\mathcal{N}^+ \to \mathcal{N}^+$ una función que caracteriza la eficiencia de un (trozo de) programa a partir del volumen de los datos de entrada. Es decir, O(f) contiene todas aquellas funciones acotadas por cualquier variante lineal de f, desdeñando los valores más pequeños de su parámetro n; así, en O(n) nos encontramos $3n$, n+5, 9000n+3456, etc., pues todas ellas encajan en la definición dando valores apropiados a c_0 y n_0. Puede resumirse esta definición diciendo que la eficiencia de un programa no depende de factores de bajo nivel de la instalación (ésto nos lo asegura la constante c_0) ni del comportamiento del programa para pequeños volúmenes de datos (ésto nos lo asegura la constante n_0).

La definición puede extenderse fácilmente al caso general en que la eficiencia de un programa depende del volumen de más de un dominio de datos [BB87]:

$$O(f) = \{g: \mathcal{N}^+ \times \ldots \times \mathcal{N}^+ \to \mathcal{N}^+ / \exists c_0, x_1, \ldots, x_k \in \mathcal{N}^+:$$
$$\forall n_1, \ldots, n_k: n_1 \geq x_1 \wedge \ldots \wedge n_k \geq x_k: g(n_1, \ldots, n_k) \leq c_0 f(n_1, \ldots, n_k)\}$$

De ahora en adelante, a los n_1, ..., n_k los llamaremos parámetros de eficiencia.

En el uso de la notación O para medir la eficiencia de programas, hay diversas funciones que aparecen con frecuencia con el significado siguiente:

$O(f) + O(g) = O(f + g) = O(\text{máximo}(f, g))$,

siendo $\text{máximo}(f, g)(n_1, \ldots, n_k) = \text{máximo}(f(n_1, \ldots, n_k), g(n_1, \ldots, n_k))$

$O(f) * O(g) = O(f * g)$, siendo $(f * g)(n_1, \ldots, n_k) = f(n_1, \ldots, n_k) \cdot g(n_1, \ldots, n_k)$

$\log(O(f)) = O(\log f)$, siendo $(\log f)(n_1, \ldots, n_k) = \log f(n_1, \ldots, n_k)$

$\text{pot}(O(f), r) = O(\text{pot}(f, r))$, siendo $(\text{pot}(f, r))(n_1, \ldots, n_k) = (f(n_1, \ldots, n_k))^r$, $r \in \mathcal{N}^+$

$\exp(r, O(f)) = O(\exp(r, f))$, siendo $(\exp(r, r))(n_1, \ldots, n_k) = r^{f(n1, \ldots, nk)}$, $r \in \mathcal{N}^+$

2.2. Expresiones válidas en la notación O

Vamos a definir las reglas de formación de expresiones válidas en la notación O (que llamamos O-expresiones) para determinar la eficiencia de los programas en nuestro enfoque. Daremos aquí una definición inductiva, y en la sección 4 introduciremos una gramática BNF para las mismas. Las O-expresiones válidas se definen sobre un conjunto $S = \{n_1, ..., n_k\}$ de parámetros de eficiencia, y son:

- $\forall i$: $1 \leq i \leq k$: n_i es una O-expresión válida.
- 1 es una O-expresión válida (representa O(1), el tiempo de ejecución constante).
- Para toda O-expresión válida E, log(E) es una O-expresión válida.
- Para toda O-expresión válida E y para todo natural positivo r, pot(E, r) y exp(r, E) son una O-expresión válida.
- Para todo par de O-expresiones válidas $E1$ y $E2$, $E1 + E2$ y $E1 * E2$ son O-expresiones válidas.
- Nada más es una O-expresión válida.

2.3. Cálculo de simplificación de O-expresiones

Los operadores presentados se caracterizan por cumplir unas relaciones que permiten manipular las O-expresiones y así simplificar los resultados del proceso de cálculo de eficiencia. Destacamos:

- 1 actúa como elemento neutro de la suma y del producto:

  $x + 1 = x$, $x * 1 = x$

- La suma y el producto son asociativos y conmutativos:

  $x + y = y + x$, $(x + y) + z = x + (y + z)$, $x * y = y * x$, $(x * y) * z = x * (y * z)$

- Simplificación de factores negligibles:

  $x + (x * y) = x * y$, $pot(x, k) + log(x) = pot(x, k)$

  $pot(x, k) * y + pot(x, j) = pot(x, k) * y$ si $k \geq j$

- Propiedades de la potencia y la exponenciación:

  $pot(1, k) = 1$, $pot(x, 1) = x$, $pot(x, k) * pot(x, j) = pot(x, k+j)$

  $exp(k, 1) = 1$, $exp(1, x) = 1$

- Propiedades del logaritmo:

  $log(1) = 1$, $log(pot(x, k)) = log(x)$

Estas igualdades deben aplicarse cuidadosamente para no caer en bucles durante el proceso de simplificación.

3. Un lenguaje de trabajo

Definimos en esta sección un pequeño lenguaje imperativo como vehículo para la formulación de nuestra propuesta. No obstante sus reducidas dimensiones, el lenguaje ya es suficientemente completo porque introduce las estructuras de control habituales y, en particular, el bucle, y también porque permite llamadas a funciones y acciones, e incluye un mecanismo de definición de tipos básicos (aunque la estructura de los tipos es en realidad irrelevante para el cálculo de la eficiencia temporal, nos permitirá mostrar un ejemplo no trivial).

Introducimos en la fig. 1 la gramática BNF del lenguaje. Destacamos los puntos siguientes:

- Vemos que un programa es una lista de instrucciones seguida de la codificación de las funciones y acciones que aparecen en ella, y precedida por la declaración de tipos y de datos necesarios para su escritura; las acciones y funciones, a su vez, declaran datos y ejecutan una lista de instrucciones, devolviendo una expresión como resultado en el caso de las funciones. Queda claro que la estructura modular del lenguaje es mejorable, pero ésto no nos preocupa porque este aspecto no influye en el cálculo del tiempo de ejecución.
- La declaración de tipos permite definir vectores y tuplas (*records*). Por lo que se refiere a los datos, no se permite declararlos de tipo anónimo.
- El repertorio de instrucciones que contemplamos es: la asignación (de datos de tipo básico: enteros, booleanos, etc.), un condicional de dos ramas, un bucle, la invocación a acción y la instrución nula. Para simplificar algunos detalles irrelevantes de cara a los objetivos del artículo, damos la misma forma sintáctica a ambas partes de la asignación.
- Las expresiones son de la forma habitual, permiendo invocaciones a funciones. No nos preocupamos de los problemas de ambiguedad de *parse* por no afectar al cálculo de la eficiencia. Supondremos que los operadores unarios y binarios se aplican sobre tipos básicos (así, no es lícito comparar dos vectores, por ejemplo).

```
<programa> ::= <tipos> <datos> <linst> <laccfnc> | <tipos> <datos> <linst>
<laccfnc> ::= <laccfnc> <acc> | <laccfnc> <fnc> | <acc> | <fnc>
<acc> ::= acción ID '(' <datos> ')' es <datos> <linst> fin
<fnc> ::= función ID '(' <datos> ')' ':' ID es <datos> <linst> devuelve <expr> fin
<tipos> ::= <tipos> ID '=' <un_tipo> | ID '=' <un_tipo>
<un_tipo> ::= <vector> | <tupla> | ID
<vector> ::= vector '[' LITERAL '..' LITERAL ']' de <un_tipo>
<tupla> ::= tupla <datos> fin
<datos> ::= <datos> ';' ID ':' <un_tipo> | ID ':' <un_tipo>
<linst> ::= <linst> ';' <inst> | <inst>
<inst> ::= <assign> | <cond> | <bucle> | <invoc_acc> | nada
<assign> ::= <expr> ':=' <expr>
<cond> ::= si <expr> entonces <linst> sino <linst> fsi
<bucle> ::= mientras <expr> hacer <linst> fmientras
<invoc_acc> ::= ID '(' <lexpr> ')'
<lexpr> ::= <lexpr> ',' <expr> | <expr>
<expr> ::= '(' <expr> ')' | OPUN <expr> | <expr> OPBIN <expr> | ID '(' <lexpr> ')' |
           <expr> '.' ID | <expr> '[' <expr> ']' | ID | LITERAL
```

Fig. 1: Una gramática para el lenguaje de trabajo.

4. Reglas básicas para el cálculo de la eficiencia

A continuación, vamos ya a dirigirnos al cálculo automático de la eficiencia de los programas escritos en el lenguaje anterior. Presentamos las construcciones que nos permitirán establecer la eficiencia de las diferentes construcciones del lenguaje, las usamos sobre nuestro lenguaje de trabajo y, finalmente, las aplicamos sobre un ejemplo concreto.

4.1. Un atributo sintetizado para el cálculo de la eficiencia

Definimos la existencia de un atributo sintetizado sobre la gramática, el tiempo asintótico de ejecución medido con la notación O, identificado por el símbolo O. El valor que tomará este atributo se podrá determinar en función de las reglas de la gramática; así, para cada regla del estilo <no terminal> ::= parte derecha, deberá escribirse el valor para el atributo correspondiente, O(<no terminal>), en función de los símbolos que componen la parte derecha.

Una primera cuestión que aparece es que, generalmente, para cada no terminal del lenguaje existen diversas reglas que lo reducen. Por lo tanto, necesitamos una notación que nos permita saber a qué regla se refiere una expresión de cálculo del valor del atributo. En concreto, dadas las reglas:

<NT> ::= r1 | r2 | ... | r*k*

es necesario establecer el tiempo de ejecución de cada una de ellas mediante:

O(<NT>-1) ::= O-expresión en función de los símbolos de r1

...

O(<NT>-*k*) ::= O-expresión en función de los símbolos de r*k*

Cualquier aparición de un símbolo terminal en la O-expresión (identificador -representando un parámetro de eficiencia- o número natural) se interpreta como una referencia a su valor.

Definimos un par de abreviaturas útiles:

- Si existen diversas reglas cuyo valor de atributo O es idéntico, pueden agruparse:

  O(<NT>-*i*1, ..., <NT>-*in*) ::= O-expresión común para las reglas *i*1-ésima, ..., *in*-ésima

- Si todas las reglas tienen el mismo valor de atributo O, pueden omitirse los sufijos:

  O(<NT>) ::= O-expresión común para todas las reglas r1, ..., r*k*

Por otro lado, si existen reglas cuyos símbolos a reducir <NT> no intervienen en el valor del atributo de ninguna otra regla de la forma <NT2> ::= ... <NT>..., puede dejarse el valor de O indefinido en ellas. Esto ocurre en reglas para construcciones que no afectan el tiempo de ejecución, como puede ser una declaración de variables.

A veces, en la parte derecha de una regla aparece un mismo símbolo (terminal o no) repetidamente. Si este símbolo interviene en el valor del atributo para la regla, se produce una ambigüedad, pues su simple escritura no identifica a cuál de las múltiples apariciones nos estamos refiriendo. Para solucionar este problema, podemos utilizar otra vez el sufijo "-*k*" que en este caso se interpreta como: "*k*-ésima aparición del símbolo referenciado en un examen de la regla de izquierda a derecha". Notemos, pues, que el sufijo tiene un significado diferente según se aplica sobre el no-terminal a reducir o sobre símbolos de la parte derecha.

Por último, existe la posibilidad de que la eficiencia de una regla dependa no sólo de la sintaxis sino de ciertos aspectos semánticos. En ocasiones, esta dificultad será insoslayable y deberá modificarse la sintaxis del lenguaje para añadir más información a los programas fuente (por ejemplo, al estudiar la estructura iterativa -v. 4.2) pero, a veces, los aspectos semánticos son más simples y la situación puede reconducirse estableciendo la eficiencia de la regla mediante parámetros. En concreto, dada una regla <NT> ::= ... T ... tal que O(<NT>) depende del valor concreto del símbolo terminal T, y tal que T también aparece en otra regla <NT2> ::= ... T ... conectada de alguna forma con <NT> y cumpliéndose que T actua de identificador de cada instancia de <NT2> en el programa, puede declararse el valor de O(<NT2>) parametrizado por T y el valor de O(<NT>) como una instancia del parámetro T de O(<NT2>):

O(<NT2>)(T) ::= valor en función de los símbolos de la parte derecha

O(<NT>) ::= en función de O(<NT2>)(T)

En el siguiente apartado aparece un ejemplo de esta situación con la invocación a acciones y funciones.

4.2. Determinación de la eficiencia del lenguaje de trabajo

Para empezar, definimos el tiempo de ejecución de un programa como el tiempo de ejecutar la lista de instrucciones que lo componen. Notemos que no debe incluirse en el resultado el tiempo de ejecución de las acciones y funciones que aparecen a continuación; este tiempo se tendrá en cuenta cada vez que aparezca una invocación. Así, pues, tenemos:

O(<programa>) ::= O(<linst>)

Aunque existan dos reglas que reducen el símbolo no terminal <programa>, como en ambas el valor del atributo es el mismo, esta expresión es suficiente.

Vamos a establecer a continuación el tiempo de ejecución de una acción o función. Como en el caso anterior, el tiempo depende del coste de la ejecución de la lista de instrucciones que la forman (para simplificar, consideramos constante el tiempo de paso de parámetros); en el caso de la función, además, debe considerarse el tiempo de evaluar la expresión final. De momento (después veremos la necesidad de proponer algún cambio), formulamos:

O(<acc>) ::= O(<linst>) O(<fnc>) ::= O(<linst>) + O(<expr>)

En cuanto a las instrucciones, debe escribirse el tiempo de ejecutar una lista de instrucciones como la suma del tiempo de ejecución de sus componentes. Además, para cada instrucción obtenemos su eficiencia:

O(<linst>-1) ::= O(<linst>) + O(<inst>) O(<linst>-2) ::= O(<inst>)

O(<inst>-1) ::= O(<assign>) O(<inst>-2) ::= O(<cond>)

O(<inst>-3) ::= O(<bucle>) O(<inst>-4) ::= O(<invoc_acc>) O(<inst>-5) ::= 1

El tiempo de la asignación, considerando que tan solo es posible asignar valores de tipo simple, queda igual al tiempo de evaluación de las expresiones que aparecen en ambas partes:

O(<assign>) ::= O(<expr-1>) + O(<expr-2>)

La regla para el condicional es similar:

O(<cond>) ::= O(<expr>) + O(<linst-1>) + O(<linst-2>)

Los problemas empiezan si consideramos ahora la invocación a acción, pues no existe ningún medio para referenciar el tiempo de ejecución de la acción invocada. Para ello, debemos modificar las reglas ya vistas anteriormente para <acc> y <fnc> añadiendo nueva información. En concreto, establecemos que el tiempo de ejecución de una invocación a acción o función es referenciable a partir del identificador, usando el mecanismo de parametrización que hemos explicado en el apartado anterior. Las reglas quedan, entonces:

O(<acc>)(ID) ::= O(<linst>) O(<fnc>)(ID) ::= O(<linst>) + O(<expr>)

O(<invoc_acc>) ::= O(<acc>)(ID) + O(<lexpr>)

O(<lexpr>-1) ::= O(<lexpr>) + O(<expr>) O(<lexpr>-2) ::= O(<expr>)

Es decir, las dos primeras reglas establecen el tiempo de ejecución de la acción o función *ID* (siendo *ID* el valor del identificador de la parte derecha de la regla), y la tercera regla establece que el tiempo de ejecución es igual al tiempo calculado para la acción referenciada más el tiempo de evaluación de las expresiones que aparecen en los parámetros.

También el bucle nos provoca un problema. En este caso, ocurre que es imposible inferir el número de veces que se ejecutará el bucle simplemente a partir de la estructura sintáctica del mismo. Por ello, es necesario que el programador añada información en el código del bucle y la mejor manera para hacerlo es incorporar esta información a la gramática del lenguaje:

<bucle> ::= **mientras** <expr> **hacer con cota** <O-expr> <linst> **fmientras**

donde <O-expr> es una O-expresión válida tal y como se ha descrito en la sección 2:

<O-expr> ::= '(' <O-expr> ')' | ID | 1 |

log '(' <O-expr> ')' | **pot** '(' <O-expr> ',' LITERAL ')' |

<O-expr> '+' <O-expr> | <O-expr> '*' <O-expr> |

Ahora, la regla del bucle puede escribirse como:

O(<bucle>) ::= (O(<expr>) + O(<linst>)) * O(<O-expr>)

Las reglas de cálculo de eficiencia de las expresiones son triviales o similares a algunas ya explicadas; consideramos constante el coste de refenciar una posición de vector (aunque el cálculo de la posición sí que puede tener un coste asociado) o un campo de tupla:

O(<expr>-1, <expr>-2, <expr>-5) ::= O(<expr>)

O(<expr>-3, <expr>-6) ::= O(<expr>-1) + O(<expr>-2)

O(<expr-4>) ::= O(<fnc>)(ID) + O(<lexpr>)

O(<expr-8>, <expr>-9) ::= 1

Finalmente, el coste asociado a las reglas de expresiones asintóticas es:

O(<O-expr>-1) ::= O(<O-expr>) O(<O-expr>-2) ::= ID

O(<O-expr>-3) ::= 1 O(<O-expr>-4) ::= log(O(<O-expr>))

O(<O-expr>-5) ::= pot(O(<O-expr>), LITERAL)

O(<O-expr>-6) ::= O(<O-expr>-1) + O(<O-expr>-2)

O(<O-expr>-7) ::= O(<O-expr>-1) * O(<O-expr>-2)

4.3 Ejemplo

Consideramos una porción de código *T* destinada a examinar todas las aristas de un grafo dirigido y etiquetado; las aristas se manipulan mediante una acción *manipular* cuya ejecución supondremos que tiene un coste O(*k*). Suponemos que el grafo presenta una operación para obtener la lista de nodos que lo forman, y otra que, dado un vértice *x*, obtiene la lista de nodos sucesores de *x* en el grafo (es decir, aquéllos unidos con *x* por una arista) junto con su etiqueta; el tiempo de ejecución de la primera es igual al número de nodos del grafo, que representamos por un parámetro de eficiencia denominado *num_nodos*; en cuanto al segundo, y considerando el peor de los casos en que existen aristas desde *x* hacia todos los otros nodos, también tenemos como complejidad O(*num_nodos*). Por lo que respecta a las listas, suponemos la existencia de las operaciones habituales para su recorrido: situarnos al inicio, obtener el actual, avanzar al siguiente y saber si se ha recorrido toda la lista; suponemos también que su coste es O(1).

```
obtener_lista_nodos(G, lnodos); posicionar(lnodos)
mientras no final?(lnodos) hacer con cota num_nodos
  obtener_sucesores(G, actual(lnodos), lsucc)
  posicionar(lsucc)
  mientras no final?(lsucc) hacer con cota num_nodos
     x := actual(lsucc); manipular(x)
     avanzar(lsucc)
  fmientras
  avanzar(lnodos)
fmientras
```

Fig. 2: Una porción de código T.

Aplicando las reglas de cálculo del atributo O, obtenemos:

$$O(T) = O(\text{<invoc\_acc>})(\text{obtener\_lista\_nodos}) + O(\text{<invoc\_acc>})(\text{posicionar}) + \\ + (O(\text{<invoc\_fnc>}(\text{final?})) + O(T')) * \text{num\_nodos},$$

siendo T' el cuerpo del bucle externo (la evaluación de los parámetros de las invocaciones a acciones y funciones que aparecen tienen coste constante y por eso ya no las mostramos). Dado el tiempo de ejecución que hemos supuesto para las acciones y funciones que aparecen, queda:

$$O(T) = \text{num\_nodos} + 1 + (1 + O(T')) * \text{num\_nodos} = \text{num\_nodos} + (O(T') * \text{num\_nodos})$$

El análisis de T' es similar al anterior, y obtenemos:

$$O(T') = \text{num\_nodos} + 1 + (1 + O(T'')) * \text{num\_nodos} + 1,$$

siendo T'' el cuerpo del bucle interno, cuyo coste es $O(1) + O(k) + O(1) = O(k)$; queda, pues:

$$O(T') = \text{num\_nodos} + k * \text{num\_nodos} = k * \text{num\_nodos}$$

y, en consecuencia:

$$O(T) = \text{num\_nodos} + (k * \text{num\_nodos} * \text{num\_nodos}) = \\ \text{num\_nodos} + (k * \text{pot}(\text{num\_nodos}, 2)) = k * \text{pot}(\text{num\_nodos}, 2)$$

que es el valor final del atributo.

Destacamos que todas las simplificaciones que hemos efectuado en este cálculo de eficiencia se basan en la aplicación de las igualdades introducidas en el apartado 2.1.

5. Esqueletos

El ejemplo de la sección anterior ilustra el problema más habitual de cálculo de la eficiencia en el peor de los casos usando unas reglas fijas: la pérdida de precisión. Supongamos que la implementación elegida para los grafos es mediante listas de adyacencia (es decir, cada nodo incluye una lista que contiene sus nodos sucesores junto con la etiqueta de la arista). En este caso, si bien es verdad que el tiempo de ejecución individual de una operación *obtener_sucesores* puede llegar a O(*num_nodos*), el tiempo de ejecución global de visitar los sucesores de todos los nodos es, en realidad, igual al número de aristas, que podemos representar por *num_aristas*. Si el grafo es denso, *num_aristas* será del orden del cuadrado del número de nodos; en cambio, si el grafo es disperso, *num_aristas* puede llegar a ser asintóticamente equivalente a *num_nodos* y, en ese caso, el valor de la eficiencia de T que hemos calculado con las reglas vistas en 4.2 está lejos de la realidad.

Para tratar este problema introducimos el concepto de esqueleto. Los esqueletos definen patrones de combinación de unas determinadas operaciones junto con su tiempo de ejecución. Sintácticamente, son trozos de programas que no sólo involucran variables, llamadas a acciones y funciones, etc., sino también símbolos no terminales que pueden ser instanciados en contextos particulares. Los esqueletos están generalmente asociados a tipos abstractos de datos concretos, y su identificación debería ser responsabilidad del especificador del tipo.

En la fig. 3 definimos un esqueleto llamado *visita_de_aristas* que consulta todas las aristas del grafo y les aplica un tratamiento determinado, que queda aquí representado por el símbolo no terminal <linst>. Se fija el tiempo de ejecución como O(*num_aristas*) multiplicado por el coste de tratar cada arista, que queda pendiente de refinar pues depende de la instancia concreta de <linst>. Notemos que los bucles no incluyen información sobre su tiempo de ejecución, pues en este caso la eficiencia no se calcula de manera automática.

El proceso de cálculo de la eficiencia es ligeramente diferente al tratar con esqueletos. Precisamente, lo primero que se busca en los programas son instancias de esqueletos, pues la eficiencia de éstos tiene prioridad sobre el cálculo de la eficiencia mediante las reglas asociadas a la gramática. La búsqueda de instancias se concreta en un proceso de *pattern matching* que involucra los árboles de *parse* asociados al programa y a los esqueletos definidos; diremos que

se ha encontrado una instancia de esqueleto cuando existe un subárbol del programa igual al esqueleto una vez se han sustituido las variables de éste por objetos del programa, y los símbolos no terminales por estructuras sintácticas que reducen a dichos símbolos. Una vez se ha identificado la instancia, su eficiencia se calcula a partir del coste asociado a los símbolos no terminales que aparecen en el esqueleto y que intervienen en el coste establecido en la cabecera del mismo.

```
esqueleto visita_de_aristas
    coste num_aristas * O(<linst>)
    var G: grafo; lnodos, lsucc: lista fvar
        obtener_lista_nodos(G, lnodos); posicionar(lnodos)
        mientras no final?(lnodos) hacer
                obtener_sucesores(G, actual(lnodos), lsucc)
                posicionar(lsucc)
                mientras no final?(lsucc) hacer
                        <linst>
                        avanzar(lsucc)
                fmientras
                avanzar(lnodos)
        fmientras
fin
```

Fig. 3: Un esqueleto para tratar las aristas de un grafo.

Por ejemplo, el trozo de programa *T* de la fig. 2 de la sección anterior es claramente una instancia del esqueleto *visita_de_aristas*, asociando al símbolo no terminal <linst> la secuencia de instrucciones "x := actual(lsucc); manipular(x)". En este caso, la eficiencia del programa se determina a partir de la eficiencia del esqueleto; ahora bien, el valor de O(<linst>) sí que se calcula según las reglas habituales, que determinan:

$$O(\text{<linst>}) = O(x := \text{actual(lsucc)}) + O(\text{<invoc\_acc>})(\text{manipular}) = 1 + k = k$$

por lo que el valor de la eficiencia de *T* es $O(T) = num\_aristas * k$. Destacamos que el valor de esta expresión es inferior al que habíamos calculado previamente cuando *num_aristas* sea (asintóticamente) menor que el cuadrado de *num_nodos*.

6. Conclusiones y trabajo futuro

Hemos presentado una propuesta para el cálculo automático de la eficiencia de programas imperativos, entendiendo por tal su tiempo de ejecución, medido con la notación asintótica O. Esta propuesta se basa en la definición de un atributo sintetizado en la gramática del lenguaje y en un sistema de formulación de patrones, que hemos denominado esqueletos, para identificar trozos de programa cuya eficiencia se establece explícitamente, de manera parametrizada. Hemos desarrollado nuestro trabajo estudiando un lenguaje de programación imperativo simple pero suficientemente completo. Como resultado, es posible determinar el tiempo de ejecución de un programa imperativo estructurado, con acciones, funciones y tipos definidos por el usuario. El precio a pagar consiste en: 1) definir las reglas de cálculo del atributo sintetizado; 2) identificar y definir los esqueletos adecuados; y 3) establecer el número máximo de veces que se ejecutará cada bucle del programa.

En cuanto al trabajo futuro, hay varias líneas principales de investigación. Primero, debemos desarrollar el primer prototipo de la herramienta. También queremos adaptar la propuesta a lenguajes de programación imperativos o orientados a objetos existentes y de amplia difusión. Para ello, ampliaremos la gramática de nuestro lenguaje de trabajo para que abarque la mayoría

de construcciones presentes en dichos lenguajes, facilitando así la posterior adaptación. Deberemos cuidar especialmente el efecto de la genericidad y de la herencia en el cálculo automático de la eficiencia.

En tercer lugar, estamos trabajando actualmente en el desarrollo de un marco formal para la propuesta, posiblemente involucrando la técnica de interpretación abstracta [CC77], que ha sido objeto de interés por parte de otros equipos [AiA93, Ros89] para trabajos similares, comprobándose su utilidad.

Finalmente, queremos conectar este trabajo con otras líneas de investigación que estamos desarrollando, sobre todo con un mecanismo de selección automática de implementaciones de componentes en base a sus características no funcionales [Fra97, Fra+97]. En particular, la eficiencia es una de las características no funcionales de los programas, y el trabajo aquí propuesto permitirá que dicha característica se calcule automáticamente, mejorando el estado actual del mecanismo que requiere la escritura explícita por parte del programador de la eficiencia de las implementaciones.

Referencias

[AiA93] Y. Aït-Ameur. "Formal Program Development by Transformation and Non Functional Properties Evaluation. An Application to Numerical Programs". *Proceedings 5th International Conference on Software Engineering and Knowledge Engineering*, 1993.

[BB87] G. Brassard, P. Bratley. *Algorithmique. Conception et Analyse*. Ed. Masson, 1987.

[Bra85] G. Brassard. "Crusade for a better Notation". SIGACT News, 16(4), 1985.

[CC77] P. Cousot, R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". *Proceedings ACM POPL*, Los Angeles (California), 1977.

[Fra97] X. Franch. "Including Non-Functional Issues in Anna/Ada Programs for Automatic Implementation Selection". *Proceedings of Ada-Europe'97. International Conference on Reliable Software Technologies*, LNCS 1251, Springer-Verlag, 1997.

[Fra+97] X. Franch, P. Botella, X. Burgués, J.M. Ribó. "ComProLab: A Component Programming Laboratory". *Proceedings 9th Software Engineering and Knowledge Engineering* (SEKE), Madrid (España), 1997.

[FSZ89] P. Flajolet, B. Salvy, P. Zimmermann. "Lambda-Upsilon-Omega: An Assistant Algorithms Analyzer". En *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, LNCS 357, Springer-Verlag, 1989.

[FSZ91] P. Flajolet, B. Salvy, P. Zimmermann. "Automatic average-case analysis of algorithms". *Theoretical Computer Science* 79, 1991.

[HC88] T. Hickey, J. Cohen. "Automating Program Analysis". *Journal ACM*, 35, 1988.

[Knu76] D. Knuth. "Big Omicron and Big Omega and Big Theta". SIGACT News, 8(2), 1976.

[LeM88] D. Le Métayer. "ACE: An Automatic Complexity Evaluator". *ACM TOPLAS* 10(2), 1988.

[Ros89] M. Rosendahl. "Automatic Complexity Analysis". En *FPCS 89*, 1989.

[Web75] B. Wegbreit. "Mechanical Program Analysis". *Communications ACM*, 18(9), 1989.

[Zim89] P. Zimmermann. "Alas: un système d'analyse algébraique". Rapport de recherche 968, INRIA, 1989.