novática

UPGRADE **Revista de la Asociación de Técnicos de Informática**

Novática, revista fundada en 1975 y decana de la prensa informática española, es el órgano oficial de expresión y formación continua de **ATI** (Asociación de Técnicos de Informática), que edita también la revista **REICIS** (Revista Española de Innovación, Calidad e Ingeniería del Software). ***Novática*** edita asimismo **UP**GRADE, revista digital de **CEPIS** (*Council of European Professional Informatics Societies*), en lengua inglesa, y es miembro fundador de **UP**ENET (**UP**GRADE *E*uropean *NET**work*)

<http://www.ati.es/novatica/>
<http://www.ati.es/reicis/>
<http://www.upgrade-cepis.org/>

ATI es miembro fundador de **CEPIS** (*Council of European Professional Informatics Societies*) y es representante de España en **IFIP** (*International Federation for Information Processing*); tiene un acuerdo de colaboración con **ACM** (*Association for Computing Machinery*), así como acuerdos de vinculación o colaboración con **AdaSpain**, **AI2** y **ASTIC.**

Consejo Editorial
Antoni Carbonell Nogueras, Juan ManuelCueva Lovelle, Juan Antonio Esteban Iriarte,Francisco López Crespo, Celestino Martín Alonso, Josep Molas i Bertrán, Olga Pallás Codina, Fernando Piera Gómez (Presidente del Consejo), Ramón Puigjaner Trepat, Miquel Sàrries Griñó, Asunción Yturbe Herranz

Coordinación Editorial
Rafael Fernández Calvo <rfcalvo@ati.es>

Composición y autoedición
Jorge Llácer Gil de Ramales
Traducciones
Grupo de Lengua e Informática de ATI <http://www.ati.es/gt/lengua-informatica/>, Dpto. de Sistemas Informáticos - Escuela Superior Politécnica - Universidad Europea de Madrid
Administración
Tomás Brunete, María José Fernández, Enric Camarero, Felicidad López

Secciones Técnicas: Coordinadores

Administración Pública electrónica
Gumersindo García Arribas, Francisco López Crespo (MAP)
<gumersindo.garcia@map.es>, <flc@ati.es>
Arquitecturas
Enrique F. Torres Moreno (Universidad de Zaragoza) <enrique.torres@unizar.es>
Jordi Tubella Morgadas (DAC-UPC) <jordi@ac.upc.es>
Auditoría SITIC
Marina Touriño Troitiño, Manuel Palao García-Suelto (ASIA)
<marinatourino@marinatourino.com>, <manuel@palao.com>
Derecho y tecnologías
Isabel Hernando Collazos (Fac. Derecho de Donostia, UPV)<ihernando@legaltek.net>
Elena Davara Fernández de Marcos (Davara & Davara) <edavara@davara.com>
Enseñanza Universitaría de la Informática
Joaquín Ezpeleta Mateo (CPS-UZAR) <ezpeleta@posta.unizar.es>
Cristóbal Pareja Flores (DSIP-UCM) <cpareja@sip.ucm.es>
Gestión del Conocimiento
Joan Baiget Solé (Cap Gemini Ernst & Young) <joan.baiget@ati.es>
Informática y Filosofía
Josep Corco Juvinyà (UIC) <jcorco@unica.edu>
Esperanza Marcos Martínez (ESCET-URJC) <cuca@escet.urjc.es>
Informática Gráfica
Miguel Chover Sellés (Universitat Jaume I de Castellón) <chover@lsi.uji.es>
Roberto Vivó Hernando (Eurographics, sección española) <rvivo@dsic.upv.es>
Ingeniería del Software
Javier Dolado Cosín (DLSI-UPV) <dolado@si.ehu.es>
Luis Fernández Sanz (PRIS-EI-UEM) <lufern@dpris.esi.uem.es>
Inteligencia Artificial
Federico Barber Sanchís, Vicente Botti Navarro (DSIC-UPV)
<{vbotti, fbarber}@dsic.upv.es>
Interacción Persona-Computador
Julio Abascal González (FI-UPV) <julio@si.ehu.es>
Jesús Lorés Vidal (Univ. de Lleida) <jesus@eup.udl.es>
Internet
Alonso Álvarez García (TID) <alonso@ati.es>
Llorenç Pagés Casas (Indra) <pages@ati.es>
Lengua e Informática
M. del Carmen Ugarte García (IBM) <cugarte@ati.es>
Lenguajes informáticos
Andrés Marín López (Univ. Carlos III) <amarin@it.uc3m.es>
J. Ángel Velázquez Itúrbide (ESCET-URJC) <a.velazquez@escet.urjc.es>
Lingüística computacional
Xavier Gómez Guinovart (Univ. de Vigo) <xgg@uvigo.es>
Manuel Palomar (Univ. de Alicante) <mpalomar@dlsi.ua.es>
Mundo estudiantil
Adolfo Vázquez Rodríguez (Rama de Estudiantes del IEEE-UCM)
<a.vazquez@ieee.org>
Profesión informática
Rafael Fernández Calvo (ATI) <rfcalvo@ati.es>
Miquel Sàrries Griñó (Ayto. de Barcelona) <msarries@ati.es>
Redes y servicios telemáticos
José Luis Marzo Lázaro (Univ. de Girona), <joseluis.marzo@udg.es>
Josep Solé Pareta (DAC-UPC) <pareta@ac.upc.es>
Seguridad
Javier Areitio Bertolín (Univ. de Deusto) <jareitio@eside.deusto.es>
Javier López Muñoz (ETSI Informática-UMA) <jlm@lcc.uma.es>
Sistemas de Tiempo Real
Alejandro Alonso Muñoz, Juan Antonio de la Puente Alfaro (DIT-UPM)
<{aalonso,jpuente}@dit.upm.es>
Software Libre
Jesús M. González Barahona, Pedro de las Heras Quirós
(GSYC-URJC) <{jgb,pheras}@gsyc.escet.urjc.es>
Tecnología de Objetos
Jesus García Molina (DIS-UM) <jmolina@correo.um.es>
Gustavo Rossi (LIFIA-UNLP, Argentina) <gustavo@sol.info.unlp.edu.ar>
Tecnologías para la Educación
Juan Manuel Dodero Beardo (UC3M) <dodero@inf.uc3m.es>
Julià Minguillón i Alfonso (UOC) <jminguillona@uoc.edu>.
Tecnologías y Empresa
Pablo Hernández Medrano (Bluemat) <pablohm@bluemat.biz>
TIC y Turismo
Andrés Aguayo Maldonado, Antonio Guevara Plaza (Univ. de Málaga)
<{aguayo, guevara}@lcc.uma.es>

Las opiniones expresadas por los autores son responsabilidad exclusiva de losmismos. ***Novática*** permite la reproducción, sin ánimo de lucro, de todos los artículos, a menos que lo impida la modalidad de © o *copyright* elegida por el autor, debiéndose en todo caso citar su procedencia y enviar a ***Novática*** un ejemplar de la publicación.

Coordinación Editorial, Redacción Central y Redacción ATI Madrid
Padilla 66, 3º, dcha., 28006 Madrid
Tlfn.914029391; fax.913093685 <novatica@ati.es>
Composición, Edición y Redacción ATI Valencia
Av. del Reino de Valencia 23, 46005 Valencia
Tlfn./fax 963330392 <secreval@ati.es>
Administración y Redacción ATI Cataluña
Ciudad de Granada 131, 08018 Barcelona
Tlfn.934125235; fax 934127713 <secregen@ati.es>
Redacción ATI Andalucía
Isaac Newton, s/n, Ed. Sadiel,
Isla Cartuja 41092 Sevilla, Tlfn./fax 954460779 <secreand@ati.es>
Redacción ATI Aragón
Lagasca 9, 3-B, 50006 Zaragoza.
Tlfn./fax 976235181 <secreara@ati.es>
Redacción ATI Asturias-Cantabria <gp-astucant@ati.es>
Redacción ATI Castilla-La Mancha <gp-clmancha@ati.es>
Suscripción y Ventas
<http://www.ati.es/novatica/interes.html>, o en ATI Cataluña o ATI Madrid
Publicidad
Padilla 66, 3º, dcha., 28006 Madrid
Tlnf.914029391; fax.913093685 <novatica.publicidad@ati.es>
Imprenta
Derra S.A., Juan de Austria 66, 08005 Barcelona.
Depósito legal: B 15.154-1975 -- ISSN: 0211-2124; CODEN NOVAEC

Portada: Antonio Crespo Foix / © ATI 2006
Diseño: Fernando Agresta / © ATI 2006

Nº 179, enero-febrero 2006, año XXXII

sumario

editorial > 02
REICIS, nueva revista de ATI
La digitalización completa de *Novática*
El XL aniversario de ATI

en resumen
La Ingeniería del Software en la práctica diaria > 03
Rafael Fernández Calvo

noticias de IFIP
Reunión del *Council* de IFIP en Palma > 04

monografía

Factores clave de éxito en Ingeniería del Software
(En colaboración con **UP**GRADE)
Editores invitados: *Luis Fernández Sanz, Juan José Cuadrado Gallego y Maya Daneva*
Presentación. La Ingeniería del Software: más allá de una visión académica > 05
Luis Fernández Sanz, Juan José Cuadrado Gallego, Maya Daneva
Análisis de la Ingeniería del Software desde una perspectiva de Ingeniería > 07
Alain Abran, Kenza Meridji
Utilizando UML™ 2.0 para resolver problemas de Ingeniería de Sistemas > 21
Ian Barnard
Aplicando la medición de software orientada a servicios para obtener indicadores de calidad en componentes de código abierto > 34
René Braungarten, Ayaz Farooq, Martin Kunz, Andreas Schmietendorf, Reiner R. Dumke
El repositorio de proyectos software ISBSG & ISO 9126: una oportunidad para medir la calidad > 41
Laila Cheikhi, Alain Abran, Luigi Buglione
El factor humano en la Ingeniería del Software > 48
Luis Fernández Sanz, María José García García

secciones técnicas

Lengua e Informática
Phishing* y *pharming > 55
Grupo de Lengua e Informática de ATI, Lista Spanglish
(Mª del Carmen Ugarte García, ed.)

Lenguages informáticos
Procesamiento de páginas web con herramientas Java y XML > 57
Mireia Ribera Turró

Referencias autorizadas > 62

sociedad de la información

Personal y transferible
Patentes de Software, situación tras el rechazo europeo > 67
Alberto Barrionuevo García

Programar es crear
Estrellas diabólicas (CUPCAM 2005, problema F, enunciado) > 73
Cristóbal Pareja Flores
Programas equivalentes (CUPCAM 2005, problema E, solución) > 74
Manuel Carro Liñares, Manuel Freire Morán

asuntos interiores

Coordinación editorial / Programación de Novática > 76
Normas de publicación para autores / Socios Institucionales > 77

Monografía del próximo número: "*Entornos visuales*"

Manuel Carro Liñares[1], Manuel Freire Morán[2]
[1]Universidad Politécnica de Madrid; [2] Universidad Autónoma de Madrid

<mcarro@fi.upm.es>, <manuel.freire@uam.es>

Programas equivalentes

El enunciado de este problema apareció en el número 178 de ***Novática*** (noviembre-diciembre 2005, p. 74). Es el problema E de los planteados en el III Concurso Universitario de la Comunidad Autónoma de Madrid (CUPCAM 2005), del que ATI fue entidad colaboradora.

En nuestro desafío se proponía determinar si un programa *P*, escrito en un lenguaje simple *L*, puede transformarse reordenando sus instrucciones de modo que resulte otro programa *P'* lexicográficamente distinto tal que para cualquier valor inicial de las variables que aparecen en ellos, *P* y *P'* lleguen al mismo estado final de las mismas. *L* tiene un solo tipo de instrucciones: asignaciones de la forma

$$Var_1 := Var_2 @ Var_3$$

donde cada Var_i es el nombre de una variable y *@* es un operador binario infijo. Los programas están escritos de forma que un nombre de variable que aparezca en la izquierda de una asignación *I* debe ser nuevo y no haber aparecido en ninguna instrucción anterior a *I* ni en la parte derecha de *I* (es decir, están en forma de *Static Single Assignment*, SSA*)*. Puede, sin embargo, aparecer en la parte derecha de cualquier instrucción que siga a *I*: el programa "`a := b @ c; a := a @ c;`" no está en forma SSA, pero "`a := b @ c; d := a @ c;`" sí lo está. Es interesante apuntar que, debido a las restricciones de los programas en forma SSA, no puede haber dos instrucciones idénticas, con lo que intercambiarlas resultará en un programa distinto.

Primera aproximación

La resolución más inmediata (conceptualmente -- no necesariamente en su implementación, como veremos) consistiría en construir un programa candidato distinto del original y comprobar si el resultado de sus ejecuciones es el mismo. Las ejecuciones de programas en el lenguaje *L* son, por fortuna, siempre finitas, dado que no hay bucles ni instrucciones de salto. Por otro lado no podemos dar un valor determinado al resultado final pues no conocemos el valor inicial de las variables ni el significado del operador *@*. Como máximo podemos derivar una expresión resumiendo el término al cual cada variable ha sido ligada. Por ejemplo, el resultado de la ejecución del segundo programa de la sección anterior es el conjunto de asignaciones {`a` ← `b @ c`, `d` ← `a @ b) @ c`}. Notemos que al no saber nada acerca de las propiedades de *@* no podemos eliminar ninguno de los paréntesis, reordenar expresiones y, en general, operar.

Esta idea tiene un problema fundamental: la cantidad de permutaciones de las instrucciones de un programa de tamaño *n* es *n!*[1] Es necesario realizar código para generar esas permutaciones, estructuras de datos para almacenarlas, evaluar los programas resultantes, y construir las expresiones resultado de su evaluación. Podemos, de todos modos, realizar una implementación en un lenguaje que nos simplifique estas tareas -- Prolog, en nuestro caso (abajo).

Lo más destacable de este código es que es, prácticamente, una narración del modo de resolución del problema: "evalúa el programa inicial, genera una permutación del mismo que sea diferente del mismo y que esté en formato SSA; evalúa esta permutación y comprueba que da el mismo resultado que el programa inicial". Como puntos a favor: el *backtracking* incluído en el mismo y la facilidad de usar estructuras de datos simbólicas hacen que la codificación sea sencilla. La evaluación del programa devuelve una lista ordenada de asignaciones de expresiones a variables:

```
?- eval([a := x @ y, c := x @ a,
d := c @ c], R).
R = [a := x @ y, c := x @ (x @ y),
d := x @ (x @ y) @ (x @ (x @ y))]
```

Así, comprobar que dos programas dan el mismo resultado se reduce a comprobar igualdad de listas. Por otro lado las estructuras de datos utilizadas son muy poco sofisticadas, lo que, unido a un algoritmo de por sí ineficiente, y que no se ha intentado optimizar, hace que el programa anterior sea inaplicable en casos medianamente grandes
.

Otra solución alternativa

Como sucede en muchos casos, la solución anterior hace más de lo necesario, un pecado que a veces se comete por exceso de constructividad. En particular no sólo responde a la pregunta de si existen reordenaciones de instrucciones que preserven la semántica del programa, sino que, en caso positivo, encuentra una de ellas. Aún más, la encuentra mediante el costosísimo proceso de intentar ciegamente diferentes ordenaciones hasta dar con una satisfactoria.

¿Cuál es la condición general para que se pueda intercambiar el orden de dos (o más) instrucciones? La respuesta viene ligada a las dependencias entre instrucciones, que inducen una *ordenación topológica* de las mismas: entre distintas instrucciones puede existir una relación de precedencia que debe

```
eqprog(Prog):-
    eval(Prog, Eval),
    permute(Prog, Prog2),
    Prog \== Prog2,
    en_ssa(Prog2),
    eval(Prog2, Eval).

eval(Prog, Eval):-
    eval_t(Prog, [], Lig),
    sort(Lig, Eval).

eval_t([], L, L).
eval_t([A := B @ C|R], I, O):-
    lookup(B, I, Bv),
    lookup(C, I, Cv),
    eval_t(R, [A := Bv @ Cv|I], O).
```

```
en_ssa(Prog):- ssa_t(Prog, []).
ssa_t([], _).
ssa_t([A := B @ C | R], T):-
   \+ member(A, T),
    ssa_t(R, [A,B,C|T]).

lookup(Var, Tbl, Exp):-
    member(Var = E, Tbl) -> E = Exp ; Exp = Var.

select(X, [X|Xs], Xs).
select(X, [Y|Xs], [Y|Ys]):- select(X, Xs, Ys).

permute([], []).
permute(L, [X|P]):-
    select(X, L, L1),
    permute(L1, P).
```

ser respetada (en el sentido de mantener una ordenación temporal en su ejecución) para no alterar la semántica del programa. Estas dependencias pueden, en general, estar claras en el código (por ejemplo, porque aparezcan variables en una instrucción que hayan recibido su valor en otra instrucción anterior) o ser menos explícitas (por ejemplo, a través de zonas de memoria apuntadas por variables o accesos a ficheros externos). En *L* se da únicamente el primer caso: existe una dependencia entre una instrucción I_i y otra I_{i+k} si y sólo si la variable a la izquierda de la asignación en I_i es alguna de las que aparecen a la derecha de la asignación I_{i+k}.

Hay que resaltar que ninguna variable puede aparecer en la parte izquierda de una asignación si está presente en la parte derecha de otra instrucción anterior, ni aparecer en la parte izquierda de dos asignaciones. Sin la última restricción sería posible escribir programas como `a := b @ c; d:= a @ d; d := e @ f`, donde la última instrucción hace que las dos anteriores sean reordenables. En *L* ninguna instrucción puede afectar la reordenabilidad de las anteriores.

A partir de estas dependencias puede construirse un grafo cuyos nodos están etiquetados con instrucciones. Una ordenación topológica de los nodos (o sea, de las instrucciones) es una secuencia que incluye a todos ellos y en la cual si hay una arista de un nodo *a* a un nodo *b*, entonces el nodo *a* aparecerá necesariamente antes que el nodo *b* en la ordenación.

En general puede haber varias ordenaciones topológicas de un grafo dado; todas ellas corresponden a programas equivalentes, en el sentido de que evaluarlos arroja el mismo resultado. Un breve programa y su grafo de dependencias aparecen en la **figura 1**.

```
I1:  a := b @ c
I2:  f := c @ a
I3:  g := a @ b
I4:  h := f @ g
```

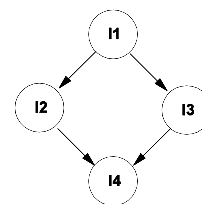

Figura 1. Programa breve y su grafo de dependencias.

Los programas $P = I_1; I_2; I_3; I_4;$ y $P' = I_1; I_3; I_2; I_4;$ son equivalentes, pues ambos son ordenaciones topológicas de nodos del grafo. Por supuesto, la instrucción I_1 debe preceder siempre a I_4 aunque no haya ninguna arista entre ambos; esto se cumple por transitividad. ¿Cuándo existe una única ordenación topológica de un grafo? Es un teorema de teoría de grafos que una ordenación topológica I_1, I_2, ..., I_n es única si y sólo si existe un arco entre cada par de nodos I_k e I_{k+1} pertenecientes a la ordenación. Es fácil de aceptar intuitivamente este teorema: en la figura anterior no hay dependencias entre las instrucciones I_2 e I_3 y pueden ejecutarse en cualquier orden. Tampoco hay relación directa entre I_1 e I_4, pero alterar el orden en que estas instrucciones se ejecutan rompería la relación de precedencia de éstas con I_2 y/o I_3.

Por tanto, lo único que debemos hacer es comprobar si en nuestro programa inicial existe una dependencia entre cada instrucción y la que le sigue. De ser así no hay ninguna reordenación alternativa que produzca el mismo resultado que el programa original. Retomando la definición de dependencia entre dos instrucciones en nuestro lenguaje, un programa no tiene equivalentes si y sólo si entre dos instrucciones sucesivas la segunda utiliza en la parte derecha de la asignación la variable que aparece en la parte izquierda de la instrucción inmediatamente anterior.

Implementación

La consideración anterior simplifica notablemente el procedimiento: sólo es necesario realizar un recorrido por el programa y no hace falta utilizar apenas ninguna estructura de datos, pues la única información que hay que mantener atañe a los nombres de variables que aparecen en dos instrucciones sucesivas. De ese modo no es necesario siquiera utilizar una tabla. Como en el enunciado (y con ánimo de simplificar la implementación) se especificó que los nombres de las variables eran de un solo carácter no es necesario trabajar con cadenas. El programa final completo queda, en C, como sigue:

```
int main(int argc, char *argv[])
{
 int nproblems;

 scanf("%\n", &nproblems);
 while(nproblems—){
  int can_be_reordered = 0;
  char LHSPrev, LHSCurr, OP1, OP2;
  int nlines;

  scanf("%d\n", &nlines);
  scanf("%c := %c @ %c;\n", &LHSCurr, &OP1, &OP2);

  if (nlines > 1) {
   while (—nlines) {
    LHSPrev = LHSCurr;
    scanf("%c := %c @ %c;\n", &LHSCurr, &OP1, &OP2);
    can_be_reordered |= ((LHSPrev != OP1) && (LHSPrev != OP2));
   }
  }
  printf(can_be_reordered ? "YES\n" : "NO\n");
 }
 return 0;
}
```

Reflexión

El análisis de independencia es esencial en muchas tareas de compilación. Quizá una de las más relevantes sea la paralelización automática: la ejecución de tareas / sentencias / etc. independientes es en general preferible, pues se minimiza la comunicación entre procesos.

De hecho, el grafo de dependencias mostrado anteriormente es una representación del esquema clásico COBEGIN-COEND de arranque de tareas paralelas. Pero la información de independencia es también útil para realizar una mejor compilación en máquinas secuenciales, pues se detectan puntos donde es posible realizar una reordenación que mejore, entre otras cosas, el *pipeline* del procesador, la localidad de acceso a las variables, etc., y que también localice posibles instrucciones redundantes.

Los analizadores reales son, por supuesto, mucho más complejos que lo que hemos apuntado aquí y deben atender a condiciones de independencia mucho más elaboradas: efectos laterales (entrada/salida, acceso a bases de datos, etc.), dependencias a partir de punteros a zonas de memoria compartidas y la posibilidad de realizar transformaciones de programas que introduzcan independencia en código que inicialmente no lo tenía, entre otras.

Nota

[1] Aunque la necesidad de que el programa candidato esté también en forma SSA puede reducir esta cantidad en gran medida.