Optimización del Orden de Evaluación de Programas Recursivos

Generalización y Extensiones de la estrategia *Programación Dinámica*

Carlos Daniel Luna[†]
Área de Computación, UNRC (Río Cuarto, Argentina). E-mail: cluna@exa.unrc.edu.ar
InCo, PEDECIBA Informática[§] (Montevideo, Uruguay). E-mail: cluna@fing.edu.uy

Gabriel Baum
LIFIA, UNLP - UNRC. E-mail: gbaum@sol.info.unlp.edu.ar

Resumen

A través de los años los científicos de la computación han identificado diversas técnicas (estrategias) generales que a menudo producen algoritmos eficientes para la resolución de muchas clases de problemas. Este trabajo presenta un análisis de la estrategia *dynamic programming* (*programación dinámica*), a partir de su formalización mediante una regla en el cálculo transformacional desarrollado en el proyecto CIP [Bauer, 85] [Bauer, 87]. La técnica especificada es utilizada en la optimización de programas recursivos ineficientes, derivados, en general, por razonamientos *divide and conquer*. A partir del análisis de la resolución de problemas sobre dominios de distinta complejidad, siguiendo un proceso sistemático y formal, se plantea una generalización de la estrategia *programación dinámica* formalizada que hace explícitas las condiciones del orden involucrado sobre el dominio de un problema. Asimismo, se formula una regla que combina la técnica de operacionalización *divide and conquer recursivo* [Grinspan, 95a] con la de optimización *programación dinámica*.

Programación dinámica evita calcular subproblemas más de una vez, invirtiendo el orden computacional y eliminando recursiones no lineales. No obstante puede conducir a que se resuelvan subproblemas innecesarios en el cómputo de ciertos problemas. Esto provoca que tanto la complejidad en tiempo como en espacio de almacenamiento resulten mayores a las deseadas. Existe una técnica conocida como *memorización* [Turner, 81] (*tabulación exacta* [Bird, 80]) que evita, por un lado, recomputar valores y por el otro, computar valores innecesarios. Sin embargo *memorización* no elimina recursiones no lineales, complicándose luego este proceso en el contexto de una metodología de desarrollo de software transformacional. Este trabajo plantea una regla que combina las ventajas de ambas técnicas, a través de un refinamiento de *programación dinámica*, en función al orden involucrado sobre el dominio del problema.

Técnicas y conceptos inherentes a la rama de *análisis de algoritmos* sirven de herramientas en la formalización y deducción de las condiciones necesarias para la utilización de las reglas propuestas. Asimismo, en la evaluación de los resultados obtenidos como consecuencia de un proceso de optimización.

Dos casos de estudio siguen el desarrollo del artículo: “cálculo de números combinatorios” y “multiplicación eficiente de *n* matrices”. Éstos destacan la relevancia de cada una de estrategias –puras, generalizadas o combinadas– que se formalizan.

Palabras claves: Estrategias de Diseño de Algoritmos: Programación Dinámica, Divide and Conquer. Especificación y Transformación de Programas. Optimización de Programas Recursivos. Análisis de Algoritmos.

[†] Deseo agradecer a Gabriel Baum (LIFIA, UNLP) por haberme incentivado y apoyado en el desarrollo de este trabajo. Asimismo a los docentes A. Pardo, L. Sierra y A. Viola del InCo (Uruguay) por las sugerencias realizadas.

[§] Becario del PEDECIBA Informática, Facultad de Ingeniería, InCo, Montevideo, Uruguay.

Optimización del Orden de Evaluación de Programas Recursivos

Generalización y Extensiones de la estrategia *Programación Dinámica*

1. Introducción

Una manera intuitiva de resolver un problema consiste en plantear una solución (especificación) declarativa, que describe la clase de algoritmos que pueden ser derivados. Con la motivación de derivar descripciones operacionales a partir de declarativas, paso a paso, se plantea el objetivo de definir reglas de transformación de programas que no sólo transformen especificaciones descriptivas en otras más operacionales, sino también operacionales en más eficientes. En [Partsch, 90] y [Grinspan, 95a] se formulan y analizan reglas que caracterizan el comportamiento de estrategias generales de solución a problemas, tales como ***divide and conquer, local search, greedy***, ***backtracking y dynamic programming***. Este trabajo desarrolla un análisis en torno a esta última estrategia y a su aplicación en la optimización del orden de evaluación de programas recursivos ineficientes, derivados, en general, por razonamientos (reglas) *divide and conquer*.

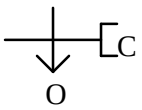
El formalismo utilizado es el cálculo transformacional definido en el proyecto CIP [Bauer, 85] [Bauer, 87]. Se trata de un lenguaje de especificaciones algebraicas de amplio espectro que contiene las construcciones clásicas de los lenguajes de programación, además de otras declarativas tales como la cuantificación universal y existencial de Primer Orden, la descripción *that* y la elección *some*, donde *some m x: P(x)* especifica la elección de algún elemento arbitrario del dominio *m* que satisfaga el predicado *P* y, *that m x: P(x)* determina al único elemento de *m* que satisface *P*. Detalles formales pueden encontrarse en [Partsch, 90]. Las nociones centrales del cálculo utilizado son *esquema de programa* y *regla de transformación*.

Un *programa* es un término bien formado sobre la signatura (Σ) de un tipo algebraico que define un lenguaje de programación a partir de algún conjunto de tipos primitivos. Un *esquema de programa* es un término sobre Σ en el que ocurren variables libres pertenecientes a un conjunto numerable X de parámetros tipados. Un esquema de programa es la generalización de un programa; los programas son esquemas de programas sin variables libres [Manna, 74]. Un ejemplo simple de esquema de programa es E(*f, x, y*), donde *f, x* e *y* son los parámetros de la expresión E. E(*f, x, y*) puede tener como instancias posibles, entre otras, los programas:

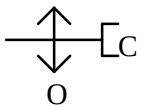
if $x < y$ then $f(x)$ else $f(y)$ fi. if $x < y$ then x else y fi.

Una *regla de transformación* es una inferencia especial que se denota de dos maneras distintas según establezca "consecuencia" o "equivalencia" entre esquemas de programas:

consecuencia



equivalencia



donde C es un conjunto de condiciones de aplicabilidad e I y O son esquemas de programas llamados 'input scheme' y 'output scheme' respectivamente.

Una regla de transformación es correcta si constituye una inferencia válida. Es decir, si se satisface C puede deducirse que O es un "descendiente" de I ($C \vdash O \subseteq I$) en el primer caso, o que O es "equivalente" a I ($C \vdash 0 \equiv I$) en el segundo. Con respecto a las condiciones de aplicabilidad, se distinguen las condiciones sintácticas de las semánticas. C alude únicamente a condiciones semánticas; para la formulación de condiciones sintácticas se utilizan predicados particulares tales como: *Kind*, *Notoccurs*, *Decl* que son introducidos a las reglas como restricciones sintácticas. Existen también predicados semánticos particulares como *Det* y *Def* [Partsch, 90].

La organización del trabajo es como sigue. La *sección 2* presenta una formalización de *programación dinámica*. La *sección 3* analiza dos casos de estudio: “cálculo de números combinatorios” y “multiplicación eficiente de secuencias de matrices”. Estos destacan la utilidad

de las distintas estrategias que se presentan. El énfasis será puesto en el análisis del segundo, por ser de mayor complejidad y a la vez más representativo de los problemas resolubles por la estrategia, de manera *optimal*. La *sección 4* desarrolla conceptos y técnicas que se relacionan a *programación dinámica*. Inicialmente expone una generalización de la regla previamente formulada que hace explícitas las condiciones del orden involucrado sobre el dominio de un problema. Luego introduce una regla que combina la técnica de operacionalización *divide and conquer recursivo* con la de optimización *programación dinámica*. Posteriormente esta sección presenta la estrategia de optimización de programas recursivos conocida como *memorización* [Turner, 81] (*tabulación exacta* [Bird, 80]). En función de las ventajas inherentes a esta técnica y las correspondientes a *programación dinámica*, se formula una regla que las combina sumando las ventajas de ambas. Finalmente, la *sección 5* expone conclusiones y trabajos futuros.

2. Programación Dinámica

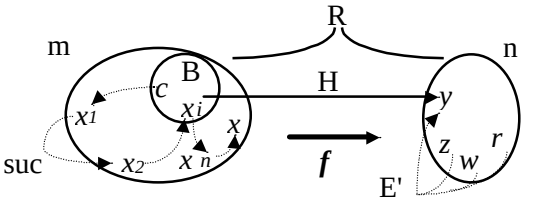
2.1 Noción intuitiva

A menudo la solución a un problema se obtiene dividiendolo en tantos subproblemas como sea necesario, y luego dividiendo cada subproblema en subproblemas más pequeños (siguiendo una metodología *divide and conquer*) produciendo un algoritmo de tiempo exponencial. No obstante, con frecuencia, sólo hay un número polinomial de subproblemas, de aquí que se deba resolver un subproblema muchas veces. Si se conserva la solución a cada subproblema resuelto, y tan sólo se toma la respuesta cuando es necesario, se obtiene un algoritmo de tiempo polinomial. Desde el punto de vista de la realización, algunas veces es más fácil crear una tabla de las soluciones de todos los subproblemas que se tengan que resolver. Se rellena la tabla sin tener en cuenta si se necesita realmente un subproblema particular en la solución total. La formación de la tabla de subproblemas para alcanzar una solución a un problema dado se denomina *programación dinámica (dynamic programming)* [Aho, 83].

2.2 Aspectos relevantes

La forma de un algoritmo de programación dinámica puede variar, pero hay un esquema común: una tabla a llenar (que implementaremos sobre un MAP) y un orden en el cual se hacen las entradas, que corresponde a un orden total y estricto con mínimo elemento definido sobre el dominio del problema. El mínimo elemento es el elemento a partir del cual se comienza a llenar la tabla; las siguientes entradas se obtienen a partir del sucesor estricto de la entrada precedente y el proceso finaliza al alcanzar el elemento cuya solución se desea calcular. El orden debe definirse de manera tal que sea posible construir la solución para un elemento a partir de la solución de algunos de sus predecesores.

2.3 Justificación intuitiva [Grinspan, 95a]



MAP

H(c)	f(x1)	f(x2)	...	f(xn)	f(x)=y

c < x 1 < x 2 < ... < x n < **x**

B(x)≡true |– f(x) = H(x)
B(x)≡false |– f(x) = E'(f,x)

m es el dominio del problema restringido a la precondición **P** y el conjunto de índices del map. ***n*** es el codominio y el tipo de los elementos del map. ***B*** es el subconjunto de elementos de ***m*** para los cuales se conoce su solución. ***R(x, y)*** es verdadero sssi ***y*** es solución de ***x*** por la función ***f***. ***suc*** es la función que dado un elemento de ***m*** retorna su sucesor estricto. ***c*** es el mínimo elemento de ***m***. ***E'*** es un esquema de programa que forma parte del cuerpo de ***f***. Además ***f*** es parámetro de ***E'***.

En un elemento de tipo map se guardan las soluciones a los subproblemas. ***map [x]*** es la solución al problema total.

2.4 Formulación de la regla. La regla presentada en [Partsch, 90] que formaliza las nociones previas es nombrada PD. LNSORD(m,<,c) especifica un orden < total y estricto –irreflexivo, transitivo y conexo– sobre *m*, con mínimo elemento *c*. E[f ' for f] denota la substitución de f por f ' en el esquema de programa E. *mode* es una notación que permite abreviar instancias de tipos.

programación dinámica
(PD)

E where
  funct f = (m x: P(x)) n:
    if B(x) then H(x) else E'(f, x) fi

LNSORD(m, <, c)
suc (x) ≡ (that m y: x < y ∧ (∀m z: x < z ⇒ y ≤ z))
DEF[f(suc(x))] |- DEF[f(x)]

E [f ′ for f] where
  funct f ′ = (m x: P(x)) n:
    tabf (c, []) [x] where
      mode tabmap = MAP(m, n, =)
      funct tabf = (m y, tabmap a: P(y) ∧ y ≤ suc(x) ∧ ∀m z: c ≤ z < y ⇒ a[z] = f(z)) tabmap:
        if y = suc(x) then a
        elsf B(y) then tabf (suc(y), a[y] ← H(y))
          else tabf (suc(y), a[y] ← E'(f, y)) fi.

3. Casos de Estudio

En esta parte del trabajo se analizará la utilidad de la regla planteada en la resolución de dos problemas representativos. El objetivo es aplicar *programación dinámica* a la optimización de programas ineficientes, siguiendo un proceso sistemático y formal [Grinspan, 95b]. El primer ejemplo refiere al cómputo del *coeficiente binomial* de dos números. El segundo a la *multiplicación "con mínimo costo" de n matrices*. Los pasos a seguir en ambos casos son:

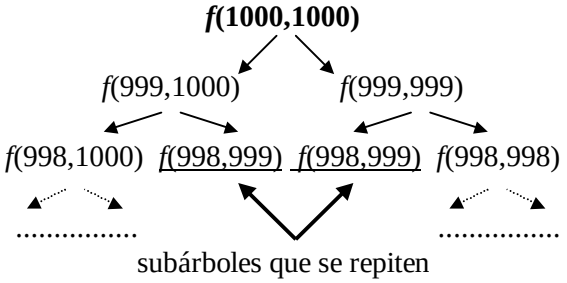
- A_1) matching sintáctico del 'input scheme' de la regla;
- A_2) determinación de las 'condiciones de diseño';
- B) instanciación y demostración de las 'condiciones de aplicabilidad'; y
- C) obtención del 'resultado por la aplicación de la regla'.

3.1 Ejemplo 1: "*Coeficiente Binomial*"

Sea *f* la función recursiva que calcula el coeficiente binomial:

funct f = (nat x, nat y: 0 ≤ y ≤ x) nat :
if y = 0 ∨ y = x then 1
else f (x-1, y) + f (x-1, y-1) fi.

Esta manera de computar el coeficiente binomial es ineficiente, puesto que en el árbol de llamadas recursivas de *f* algunos subárboles se repiten muchas veces. Esto puede verse claramente analizando la evaluación de un llamado concreto a *f*. Por ejemplo:



subárboles que se repiten

Consecuentemente, intentaremos aplicar la regla PD para optimizar el proceso de cómputo del coeficiente binomial, en cuanto al tiempo de ejecución.

A_1) Matching sintáctico del 'input scheme' de la regla PD:

nat × nat	por	m
nat	por	n
($0 \le y \le x$)	por	P
($y = 0 \vee y = x$)	por	B(x, y)
H(x, y) = 1		

A_2) Decisiones de diseño:

c = (0, 0)
(x ,y) < (i, j) ⇔ suc (x,y) = (i, j) ∨ suc (x,y) < (i, j)
funct suc (nat n, nat m: $0 \le m \le n$)) nat × nat:
  if ($m < n$) then (n, m+1) else (n+1, 0) fi.

B) Es fácil demostrar que las condiciones de aplicabilidad se satisfacen según las instancias precedentes.

C) Ahora, aplicando la regla, substituimos f por f ' resultando:

funct f ' = (nat x, nat y: $0 \le y \le x$) nat:
  tabf ((0, 0), [ ]) [x, y] where
    mode tabmap = MAP (nat × nat, nat, =);
    funct tabf = **(** nat × nat (i, j), tabmap a: $0 \le j \le i$) ∧ (i, j) ≤ suc (x, y) ∧
      ∧ ∀ nat × nat ($z1$, $z2$): (0, 0) ≤ ($z1$, $z2$) < (i, j): a[$z1$, $z2$] = f ($z1$, $z2$) **)** tabmap:
      if (i, j) = suc (x, y) then a
        elsf ($j = 0 \vee j = i$) then tabf (suc (i, j), a[i, j] ← 1)
          else tabf (suc (i, j), a[i, j] ← f (i-1, j) + f (i-1, j-1)) fi.

Observando la precondición de tabf, vemos que: f(i-1,j) = a[i-1, j] y f(i-1, j-1) = a[i-1, j-1] pues de acuerdo al orden definido (0, 0) ≤ (i-1, j) < (i, j) y (0,0) ≤ (i-1, j-1) < (i, j). Luego, la última línea de la función tabf puede substituirse por: *else tabf (suc (i, j), a[i, j]← a[i-1, j] + a[i-1, j-1]) fi*.

3.2 Ejemplo 2: "***Multiplicación eficiente de n de Matrices***"

Se tienen n matrices M_1 , ... , M_n de índices naturales y elementos de tipo r. Consideraremos definidas las funciones *filas (Mi)* y *columnas (Mi)*, que retornan el número de filas y columnas, respectivamente, de la matriz M_i. Llamaremos $M_{i\,..\,j}$ a la matriz resultado del producto M_i*...*M_j, con i ≤ j. El costo de calcular $M_{i\,..\,i+1}$ es el número de productos necesarios, es decir,

costo ($M_{i\,..\,i+1}$) = filas (M_i) * columnas (M_i) * columnas (M_{i+1}), siendo columnas (M_i) = filas (M_{i+1}).

El costo de computar $M_{i..j}$ (siendo j > i+1) depende de la asociación de las matrices M_i, ..., M_j que se utilice en el cálculo, pues el producto de matrices es asociativo y por lo tanto para determinar $M_{i..j}$ se puede elegir la asociación que involucre menos operaciones. Por ejemplo, sean las matrices M_1 de dimensión 10 por 20, M_2 de 20 por 30 y M_3 de 30 por 30; resulta:

costo (M_1 * (M_2 * M_3)) = 20×30×30 + 10×20×30 = 24000; mientras que,
costo ((M_1 * M_2) * M_3) = 10×20×30 + 10×30×30 = 15000.

Esto es, la segunda asociación realiza el 37,5% menos de productos elementales que la primera asociación, para realizar la multiplicación de las 3 matrices. Es la asociación óptima respecto al costo definido.

Luego, ***se desea especificar un algoritmo para calcular el mínimo costo de obtener $M_{1..n}$.***

Este es un problema de optimización que resulta interesante resolver, debido a que para una cantidad relativamente importante de matrices (n "grande"), la asociación ideal incurrirá en una disminución significativa de la cantidad de productos elementales necesarios, respecto de una asociación cualquiera[1]. Siendo en cualquier caso n-1 productos matriciales necesarios para n

[1] La observación tiene sentido si las matrices no son cuadradas, ya que si lo son todas las asociaciones tendrán en el

matrices. Una especificación matemática del problema es:

costo_min ($M_{1..n}$) *= min* ***{*** *costo($M_{1..i} * M_{i+1..n}$) / $1 \le i < n$* ***}****, con:*
costo_min($M_{i..i}$) = 0, $1 \le i \le n$. Donde, si $1 \le i \le j < k \le n$ y además, columnas(M_j) = filas(M_{j+1}):
costo ($M_{i..j} * M_{j+1..k}$) = costo_min($M_{i..j}$)+costo_min($M_{j+1..k}$)+ filas(M_i)×columnas(M_j)×columnas(M_k)

A continuación se plantea la formalización de una primera solución algorítmica del problema, en el formalismo elegido.

```
mode matriz = PMAP ((nat, nat), r, =);   {PMAP es el TAD map de   nat × nat   en   r}
mode sequmatriz = EESEQU (matriz ,=);  {EESEQU es el TAD secuencia de tipo matriz}

funct f = ( sequmatriz z : z ≠ λ  ∧  ∀ nat i : 1 ≤ i < |z| ⇒ columnas (z[i]) = filas (z[i+1]) ) nat :
    if |z| ≤ 1   then 0
                 else g (z, ∞, 1) fi   where

    funct g = ( sequmatriz z, nat cm, nat k ) nat :
        if |z| = k   then cm
                     else g (z, min(cm, c1+c2+c3), k+1) fi   where

        c1 = f (z[1: k])
        c2 = f (z[k+1: |z|])
        c3 = filas (z[1]) * columnas (z[k]) * columnas (z[|z|])
        funct min = (nat i, j) nat : if i ≤ j then i else j fi
```

x[i : j] denota a la subsecuencia de *x* entre los índices i y j, con $1 \le i \le j \le |x|$. Iterando con k = 1... n, la función *g* calcula el costo de $(M_{1..k})*(M_{k+1} .. n)$, registrando en *cm* el mínimo costo de las distintas asociaciones. Naturalmente, *cm* debe ser inicializado en un valor muy grande (∞). La expresión: "$z \neq \lambda \wedge \forall$ nat i : $1 \le i < |z| \Rightarrow$ columnas (z[i]) = filas (z[i+1])" es la precondición de *f*.

La solución anterior si bien puede resultar natural es ineficiente, pues su tiempo de ejecución es exponencial. Sin embargo, basta considerar el árbol de llamadas recursivas de *f* en una entrada concreta, para notar que algunos subproblemas se resuelven muchas veces. Si definimos a Ne(n) como: "la cantidad de nodos externos en el árbol de cómputo de la solución para un problema de tamaño *n*", entonces para el algoritmo anterior resulta en la siguiente ecuación recursiva:

Ne(n) = $\sum$i: $1 \le i < n$ { Ne(i) + Ne(n-i)} $2 \le n$. Con,
Ne(0) = 0 –la precondición excluye a la secuencia vacía– y,
Ne(1) = 1 –para secuencias unitarias, la solución se obtiene en un solo paso.

Luego, Ne(0) = 0, Ne(1) = 1 y $Ne(n) = 2 \times 3^{n-2}$, para $n \ge 2$. La cantidad de nodos internos (Ni) en el árbol de cómputo para el algoritmo en análisis es: $Ni(n) = 3^{n-2}$ ($2 \le n$); y, Ni(0) = Ni(1) = 0. Consecuentemente, el tiempo de ejecución del programa es $T(n) = C \times (Ne(n) + Ni(n))$, con C una constante. Esto es, T(n) es $O(3^n)$. El *peor caso* y el *caso promedio* coinciden para el algoritmo en análisis, ya que éste es independiente de la aleatoriedad de la entrada: se consideran todas las posibles asociaciones para obtener la de costo óptimo.

Sin embargo, no existe un número exponencial de secuencias contenidas Ns (subsecuencias) en una secuencia de tamaño n, sino más bien un número polinomial de ellas. Ns puede formalizarse de la siguiente manera: para cada i entre 1 y n calculamos la cantidad de prefijos de la secuencia de tamaño n-i. Es decir, Ns(n) = $\sum$i: $1 \le i \le n$ {$\sum$j: $i \le j \le n$ (1)} = $n \times (n+1) / 2$. Notemos que no se incluye a la secuencia vacía (λ) como subsecuencia, ya que la precondición la excluye. Es claro que el número de subsecuencias es una cota superior de todas las posibles secuencias *diferentes* que podrían intervenir en el cálculo de una secuencia de tamaño n, para el algoritmo en análisis. No obstante, su complejidad (orden O) es una cota estrictamente inferior para la complejidad en tiempo de cualquier solución al problema. Analizaremos posteriormente esto último con más detalle, especificando un *lower* y un *upper bound* sobre el problema.

Para optimizar el proceso de cómputo de *f* intentaremos aplicar la regla PD y obtener un

---

mismo costo. Cuanto mayor sea la disparidad entre las dimensiones de las matrices, mayor será la diferencia de costo entre las disitintas asociaciones.

algoritmo de tiempo polinomial. Primero notemos que una de las condiciones de aplicabilidad de dicha regla exige la existencia de un orden total estricto con mínimo elemento definido sobre el dominio del problema, que permita construir la solución de un elemento a partir de las soluciones de algunos de sus predecesores. Sin embargo, no podemos definir tal orden sobre *sequmatriz*. Lo que haremos entonces es considerar para cada secuencia *x* de matrices que cumpla la precondición de *f* (las matrices consecutivas de *x* deben ser multiplicables) el subconjunto m_x de sequmatriz de todas sus subsecuencias, excepto la vacía –todos los posibles subproblemas del problema *x*– y definiremos sobre m_x un orden total estricto con mínimo elemento, como veremos ulteriormente. Observemos también que cada entrada inicial *x* de *f* determina unívocamente su subconjunto m_x, y además como *f* restringida a m_x (f/m_x) es una función bien definida para todas sus entradas, podemos aplicar la regla PD a f/m_x.

A_1) Matching sintáctico del 'input scheme' de la regla PD:

Instancia	Expresión sintáctica del 'input scheme'de la regla PD
mode matriz = PMAP ((nat, nat), r, =) mode sequmatriz = EESEQU (matriz ,=);	E
(sequmatriz z : $z \subseteq x \ \wedge\ x \neq \lambda \ \wedge\ z \neq \lambda \ \wedge$ $\wedge\ \forall$ nat i : $1 \leq i < \|x\| \Rightarrow$ columnas (x[i]) = filas (x[i+1]))	m_x (con *x* inicial fijo)
Nat	n
True	P(x)
($\|z\| \leq 1$)	B(x)
0	H(x)
g (z, ∞, 1) where funct *g* = (sequmatriz z, nat cm, nat k) nat : if $\|z\| = k$ then cm else *g* (z, min(cm, c1+c2+costo), k+1) fi where c1 = *f* (z[1: k]) c2 = *f* (z[k+1: $\|z\|$]) costo = filas (z[1]) * columnas (z[k]) * columnas (z[$\|z\|$]) funct min = (nat i, j) nat : if $i \leq j$ then i else j fi	E'(f, x)

A_2) Decisiones de diseño.

Como vimos en el ejemplo 1, la dificultad en la aplicación de la regla PD reside en la determinación del orden necesario. Este punto es una *decisión de diseño* que depende, mayormente, del problema en particular abordado. Sin embargo, al analizar el dominio del problema con nociones de *análisis de algoritmos*, pueden surgir "pistas" del orden buscado. Volviendo a nuestro problema en estudio, la formalización dada de la cantidad de subsecuencias de una secuencia *x* es una manera de representar el dominio, para una función que opere sobre sus subsecuencias. El conteo de las mismas nos proporciona directamente un orden en el dominio m_x, para *x* inicial fijo. Establece una biyección del conjunto finito m_x a un subconjunto de números naturales.

$T(n) = \sum i{:}1 \leq i \leq n \{\sum j{:}\ i \leq j \leq n\ (1)\} \Rightarrow$ interpretando a *i* y a j como extremos de las subsecuencias:

x[1:1], *x*[1:2], *x*[1:3], ..., *x*[1:n],
x[2:2], ..., *x*[2:n],
. . . . . .
x[n-1:n-1], *x*[n-1:n],
x[n:n]

Teniendo en cuenta que *x*[1:n] es el problema a computar, el orden para aplicar la regla PD sería el inverso al descripto: *x*[n:n], *x*[n-1:n-1], *x*[n-1:n], ..., *x*[1:1], *x*[1:2], ..., *x*[1:n]=*x*. Luego, definimos recursivamente la relación binaria $\angle$ sobre m, de manera tal que los parámetros de las llamadas recursivas de *f* precedan en este orden, en cada caso, a su parámetro actual.

$x[i : j] \angle x[k : l] \Leftrightarrow$def. suc ($x[i : j]$) = $x[k : l] \vee$ suc ($x[i : j]$) $\angle x[k : l]$, siempre que $1 \leq i, j, k, l \leq |x|$

Donde, funct suc (m_x x[i : j]) m_x:
if (i = 1 ∧ j = |x|) then λ
elsf (j = |x|) then x[i-1: i-1]
else x[i : j+1] fi.

c = x[lxl:lxl] es el mínimo elemento. También en este caso existe un último elemento, aunque la regla no lo exige, suc(x[1:lxl]) = λ. En base a la definición de *suc* para x sobre m_x, consideramos m_x* = $m_x \cup \{\lambda\}$ (todas las subsecuencias de x), con la extensión f(λ) = 0 y *costo_min(λ)* = 0 en la especificación matemática. Formalmente el perfil de *suc* debería ser: suc (m_x x , nat i, j) m_x*; la notación x[i : j] se usa por simplicidad.

B) Demostraremos ahora que las condiciones de aplicabilidad se satisfacen según las instancias precedentes.

1) ¿LNSORD (m_x, <, x[lxl:lxl])? Es fácil demostrar que la relación ∠ define un orden total estricto sobre m_x (y sobre m_x*) con mínimo elemento x[lxl : lxl]. El dominio es finito para cada secuencia x.

2) ¿suc(z) ≡ (that m_x* y : $z \angle y \wedge \forall m_x$ w: ($z \angle w \Rightarrow y \trianglelefteq w$))? Sabemos por definición de ∠ que $z \angle$ suc(z), con lo cual probamos que suc(z) satisface la primera condición. Sea w un elemento de m_x tal que $z \angle w$ entonces, por definición de ∠, suc(z) = $w \vee$ suc(z) ∠ w, que equivale a la segunda condición: suc(z) $\trianglelefteq w$. Luego, puesto que suc(z) es único, está definido como lo requiere la condición 2.

3) ¿DEF [f(suc(z))] |– DEF [f(z)]? En efecto, f está definida sobre todos los elementos de m_x.

C) Aplicando la regla, substituimos f por f ', resultando:

```
funct f ' = (mx x) nat :
   tabf (x[lxl : lxl] ) [ x[1 : lxl] ] where
mode tabmap = MAP ( mx, nat, = );
funct tabf  = ( mx* y, tabmap a: y ⊴ suc(x)  ∧  ∀mx z: ( x[lxl : lxl] ⊴ z ∠ y  ⇒  a[z] = f (z)) ) tabmap:
            if y = suc (x)  then a
               elsf  |y| ≤ 1 then tabf (suc (y), a[y] ← 0)
                             else tabf (suc (y), a[y] ← g (y, ∞, 1) ) fi   where
             funct g = (mx z, nat cm, nat k) nat :
                if |z| = k  then cm
                            else g (z, min(cm, c1+c2+c3), k+1) fi   where
                             c1 = f (z[1: k])
                             c2 = f (z[k+1: |z|])
                             c3 = filas (z[1]) * columnas (z[k]) * columnas (z[|z|])
                             funct min = (nat i, j) nat : if i ≤ j then i else j fi
```

Observando detenidamente la precondición de tabf vemos que para toda subsecuencia z de x vale: f (z[1: k]) = a[z[1: k]] y f (z[k+1: |z|]) = a[z[k+1: |z|]], puesto que para cualquier secuencia y = x[i : j], las subsecuencias $z1$ = x[i : k] y $z2$ = x[k+1: j], con $1 \le i \le k < j \le |x|$, satisfacen:

(C_1) x[lxl : lxl] $\trianglelefteq$ $z1$ ∠ y. La primera desigualdad se establece en virtud de la minimidad de x[lxl : lxl]. La segunda se obtiene de la definición de ∠; en efecto, de acuerdo con la definición de *suc*, aplicando j-k veces *suc* a z1 (suc^{j-k}($z1$)) obtenemos la secuencia y.

(C_2) x[lxl : lxl] $\trianglelefteq$ $z2$ ∠ y. Nuevamente, la primera desigualdad se establece en virtud de la minimidad de x[lxl : lxl]. La segunda puede deducirse de la definición de ∠ ya que, según la definición de *suc*, suc^a ($z2$) = y, con a = (k-i+1)*(|x|+1) - i - (k*(k+1) - i*(i+1))/2. Gráficamente se observa en el análisis realizado para la deducción del orden (inciso A_2).

Por lo establecido precedentemente, las asignaciones c1 = f (z[1 : k]) y c2 = f (z[k+1 : |z|]) del cuerpo de la función g pueden substituirse por **c1 = a[z [1: k]] y c2 = a[z [k+1: |z|]]**, evitando recalcular subproblemas y obteniendo como deseábamos un algoritmo de tiempo polinomial, de complejidad proporcional a la suma de las longitudes de las subsecuencias de la secuencia original. Esto es, formalmente, $T(n) = C_1 + \sum i: 1 \le i \le n \{ C_2 + \sum j: i \le j \le n (j-i+1) \times C_3 \}$, que es $O(n^3)$, siendo n el número de matrices.

3.2.1 Análisis de la solución obtenida

La solución obtenida es más eficiente en tiempo de ejecución que la inicial. El dominio m_x de f/m_x, para una entrada x, coincide con el conjunto de las subsecuencias necesarias para computar la solución de x. Para calcular $x[1:n]$ con n = lxl, se deben determinar: $x[1:1]$, $x[1:2]$, ..., $x[1:n-1]$ y además, entre otras, a la siguiente subsecuencia $x[2:n]$; luego, siguiendo un razonamiento inductivo a partir de $x[2:n]$, obtenemos que se computan todas las subsecuencias de x. En consecuencia, como el nuevo algoritmo no computa subproblemas innecesarios ni calcula más de una vez un subproblema, la diferencia: $N(n) - \sum i: 1 \le i \le n \{\sum j: i \le j \le n (j-i+1)\}$ es la cantidad exacta de problemas re-computados por el primer algoritmo (nodos repetidos en el árbol de cómputo). $\sum i: 1 \le i \le n \{\sum j: i \le j \le n (j-i+1)\}$ es la cantidad total de nodos en el árbol de cómputo (que es una lista) de la solución para un problema de tamaño n, en el algoritmo obtenido con PD. El orden de $\sum i: 1 \le i \le n \{\sum j: i \le j \le n (j-i+1)\}$ resulta un *lower bound* para la complejidad de cualquier algoritmo que resuelva el problema en estudio para n matrices[2]. Esto se sigue de la especificación, del análisis de la mínima cantidad de los elementos involucrados en el cálculo del problema, subsecuencias de una dada, y de la necesidad de sus evaluaciones, propocionales a las dimensiones de las subsecuencias. Un *upper bound* intríseco al problema se sigue del siguiente análisis. Sea Na(n) el número de asociaciones posibles para una secuencia de n matrices. Na(n) cumple la siguiente definición recursiva:

Na(1) = 1 "para una secuencia de tamaño 1 hay una única asociación: [x1] $\Rightarrow$ (x1)".

Na(n) = $\sum i: 1 \le i < n \{Na(i) \times Na(n-i)\}$, para $2 \le n$ "para cada i entre 1 y n-1, todas las combinaciones entre las asociaciones posibles de [1 : i] y de [i+1 : n]".

Estas ecuaciones tienen por solución a los números de *Catalan*: $Na(n+1) = (2 \times n)! / \{n! \times (n+1!)\}$, para $1 \le n$. Luego, $Na(n+1) \approx 4^n / \{n \times (\pi \times n)^{1/2}\}$ [Sedgewick, 96]. Esto es, la complejidad inherente al problema tiene como *upper bound* $O(4^n)$, ya que la evaluación del costo de una asociación es O(n) (n-1 productos matriciales). Ambos algoritmos formulados son obviamente más eficientes, pero el obtenido por aplicación de la regla PD resulta optimal –respecto del orden $O(n^3)$ que coincide con la complejidad del lower bound del problema.

En conclusión, obtuvimos, a través de un desarrollo y análisis formales, un algoritmo optimal en tiempo de ejecución para el problema de optimizar el costo (en función al número de productos elementales necesarios) de multiplicar n matrices. Si se deseara obtener la asociación de matrices que provoca mínimo costo en la multiplicación, deberíamos reemplazar el sort *nat* por el sort par *<nat, arb_nat>*. Donde, *arb_nat* representa al TAD árbol binario de matrices con información sólo en los nodos externos (los nodos internos no guardan información explícita, sino demarcan las asociaciones). Consiguientemente la modificación del algoritmo resulta natural. El orden de tiempo de ejecución del algoritmo no se vería alterado, aunque sí, obviamente, el costo de almacenamiento sobre el MAP. La cantidad de elementos necesarios del map para una entrada de tamaño n es proporcional al número de sus subsecuencias y al costo asociado a la información almacenada para cada una de ellas (un par en la extensión porpuesta <nat, arb_nat>).

4. Generalización y Extensiones de la regla PD

Esta sección presenta una generalización y dos extensiones de la estrategia *programación dinámica*, formalizada en la regla PD. La generalización hace explícitas las condiciones del orden involucrado sobre el dominio de un problema. Las extensiones combinan la nueva regla PD obtenida con la técnica de operacionalización *divide and conquer recursivo* [Grinspan, 95a] y la de optimización *memorización* [Turner, 81] (*tabulación exacta* [Bird, 80]).

---

[2] Es la cantidad de nodos del DAG que representa al árbol de cómputo, sin elementos repetidos.

4.1 Generalización de la regla PD

El esquema de programa E' en el *input scheme* de la regla PD (presentada en 2.4) debería ser una expresión en la cual, los parámetros con los que se llama recursivamente a la función –los subproblemas–, sean predecesores estrictos del valor a calcular. Éste es el caso general impuesto al orden buscado entre los elementos del dominio (ambos ejemplos de la sección 3 así lo resaltan). Luego, resulta deseable obtener una formulación más precisa de la regla PD, en donde se haga explícita la propiedad esencial buscada de tal orden.

Sean K_1, ..., K_p funciones que determinan los valores (parámetros) de las llamadas recursivas a una función *f*, con p≥2 el caso de interés. El *input scheme* de la regla PD puede especializarse como sigue: *E* <u>*where*</u>

funct f = (m x: P(x)) n:
if B(x) then H(x) else E'(x, f(K_1(x)), ..., f(K_p(x))

Esto es, haciendo *E'(f, x) = E'(x, f(K_1(x)), ..., f(K_p(x))* en la regla PD. Consecuentemente, dos restricciones se incorporan a las condiciones de aplicabilidad de la regla:

5) P(x) Δ ¬B(x) |- c ≤ (K_i(x)) < x, ∀i: 1≤i≤p ∧ p≥2 "cada subproblema debe resultar menor en el orden buscado, para los elementos no *fáciles* (B(x) ≡ false): el orden respeta la relación nodo-padre en al árbol de cómputo, para cada problema y subproblema". Δ es la conjunción secuencial.

6) B(c) ≡ true "el elemento mínimo puede calcularse de manera directa, siendo su valor H(c)".

Luego de la generalización previa, la función **tabf** en el *output scheme*, resulta:

funct tbaf = (m y, tabmap a:P(y) ∧ y ≤ suc(x)) tabmap:
if y = suc(x) then a
elsf B(y) then tabf(suc(y), a[y] = H(y)
else tabf (suc(y), a[y] ← E'(y, a[K_1(y)], ..., a[K_p(y)])) fi.

Por último, debemos considerar la siguiente restricción sintáctica sobre cada K_i :
syntactic constrains: KIND [K_i (x)] = funct K_i (m) m, ∀i: 1 ≤ i ≤ p y p≥2.

En el ejemplo 1, K_1(x, y) = (x-1, y); K_2(x, y) = (x-1, y-1); K_1(x, y) < (x, y); K_2(x, y) < (x, y); c = (0, 0) y B(c) ≡ true. Esto es, las condiciones 5 y 6 se satisfacen.

En el ejemplo 2, $K_{1,q}$(*x*[i : j]) = [i : q]; $K_{2,q}$(*x*[i : j]) = [q+1 : j], con q∈{i, i+1, ..., j-1}; $K_{1,q}$(*x*[i : j]) < *x*[i : j]; $K_{2,q}$(*x*[i : j]) < *x*[i : j]; c = *x*[|*x*| : |*x*|] y B(c) ≡ true. Observemos que si bien el número de las funciones K varía sobre las dimensiones de las secuencias, se verifican 5 y 6 para la definición previa.

Finalmente, la generalización propuesta de la regla PD verifica, para cada *i* y *x*:
P(suc(x)) Δ ∀m z: c < z ≤ suc(x) |- isdef(a, K_i (z)) ∧ a[k_i(z)] = f(k_i(z)). Siendo *a* el map de almacenamiento e *isdef* un predicado que vale si un map está definido para un elemento dado.

4.2 Combinación de PD y *divide and conquer*

Programación dinámica es una técnica que permite mejorar la complejidad en tiempo de ejecución de ciertos programas recursivos ineficientes, obtenidos, generalmente, mediante una estrategia de diseño *divide and conquer*. Esta última técnica de diseño de algoritmos nos permite operacionalizar especificaciones descriptivas [Grinspan, 95a], mientras que la primera es una estrategia de optimización en el plano operacional. A continuación se formula una regla que combina ambos procesos de diseño en un único paso.

divide and conquer + programación dinámica

(DC&PD)

E where
  funct f = (m x: P(x)) n:
    some n y: R(x,y)

1) P(x) Δ B(x) |- R(x, H(x))
2) P(x) Δ ¬B(x) |- ∃n z_1, ..., z_p: R(K_1(x), z_1) ∧ ... ∧ R(K_p(x), z_p) ⇒
   ⇒ R(x, C'(z_1, ..., z_p))
3) P(x) Δ ¬B(x) Δ ∃ n y: R(x, y) |- ∃ n z_1, ..., z_p:
   R(K_1(x), z_1) ∧ ... ∧ R(K_p(x), z_p)
4) LNSORD (m, <, c)
5) suc(x) ≡ (that m x': x < x' ∧ (∀m z: x < z ⇒ x' ≤ z))
6) P(x) Δ ¬B(x) |- c ≤ (K_i(x)) < x, ∀i: 1≤i≤p ∧ p≥2
7) B(c) ≡ true
8) {DEF[f(x)] ⇔ DEF[g(x)]} ∧ {DEF[f(suc(x))] |- DEF[f(x)]}

E [g for f]
  funct g = (m x: P(x)) n:
    tabf(c, []) [x] where
      mode tabmap = MAP(m, n, =)
      funct tabf = (m y, tabmap a: P(y) ∧ y ≤ suc(x)) tabmap:
        if y = suc(x) then a
        elsf B(y) then tabf (suc(y), a[y] ← H(y))
          else tabf (suc(y), a[y] ← C'(a[K_1(x)], ..., a[K_p(x)])) fi.

La regla DC&PD se fundamenta en la derivación de la regla DC y en la validación de la regla PD [Grinspan, 95a]. A |- B es una notación simplificada para: A ≡ true |- B ≡ true. Sobre el dominio del problema, restringido a la precondición P, consideramos una partición en relación a problemas "fáciles" (B(x) ≡ true)[3], cuya solución se obtiene de manera directa con una función H, y problemas "no fáciles" (B(x) ≡ false). En éstos la solución se obtiene descomponiendo el problema en subproblemas (K_1, ..., K_p, con p≥2) y componiendo luego sus soluciones (C'), siguiendo un orden de cómputo inverso al "natural" (LNSORD(m,<,c)). Asimismo, se incorporan las dos condiciones analizadas anteriormente al orden sobre el dominio (condiciones 6 y 7), la restricción por la cual *f* y *g* deben estar definidas en los mismos puntos (inherente a DC), y además que *f* lo esté en todos los elementos predecesores a cada punto donde está definida.

Es importante remarcar que esta operacionalización es relevante si en los árboles de descomposición de los problemas, determinados por los K_i elegidos, se repiten subproblemas. Este análisis puede hacerse con técnicas de *análisis de algoritmos* como lo sugiere la sección 3, fundamentalmente en 3.2.

4.3 Optimización de PD: *"tabulación exacta"*

Del análisis realizado en las secciones previas se desprende que *programación dinámica* evita calcular un subproblema más de una vez, invirtiendo el orden computacional y eliminando de esta manera recursiones no lineales. Sin embargo, puede conducir a que se resuelvan subproblemas innecesarios para el cómputo del problema principal. En el ejemplo 1 de la sección 3, para computar *f*(3,2) se calculan los valores para los siguientes pares: (0,0), (1,0), (1,1), (2,0), (2,1), (2,2), (3,0), (3,1) y (3,2). No obstante, sólo resultan necesarios los subproblemas (1,0), (1,1), (2,1), (2,2) y (3,2). Esto provoca que tanto la complejidad en tiempo como en espacio de almacenamiento resulten mayores a las deseables o estrictamente necesarias.

Existe una técnica conocida como *memorización* (*tabulación exacta*) que evita, por un lado, recomputar valores y por el otro, computar valores innecesarios. Sin embargo, no elimina

[3] En general, los problemas "fáciles" son aquellos estructuralmente pequeños (términos simples del TAD que define los elementos del dominio). En el ejemplo 2 (subsección 3.2), las secuencias de a lo sumo un elemento.

recursiones no lineales, complicandose luego este proceso en el contexto de una metodología de desarrollo de software transformacional[4]. Memorización mantiene el orden computacional de cálculo de la función original y tabula los valores que son computados, asumiendo estos valores almacenados frente a la necesidad de un recomputo. La idea subyacente consiste en transformar el árbol de cómputo de la solución a un problema en un DAG, donde los nodos con más de un arco de entrada correspondan a recomputos (nodos repetidos en el árbol original). Una definición en el formalismo introducido, se obtiene como sigue:

memorización (tabulación exacta)

(Mem)

f(E) where
  funct f = (m x: P(x)) n:
        if B(x) then H(x) else E'(x, f(K_1(x)), f(K_2(x))) fi.
  Constant (E)
g(E, []) [E] where
mode map = MAP(m, n, =);
funct g = (m x, map a: P(x)) map:
  isdef(a, x) then a
        elsf B(x) then a[x] ← H(x)
              else a'[x] ← E'**(**x**,** a'[K_1(x)]**,** a'[K_2(x)]**)** fi.
where map a' = g**(**K_2(x), g(K_1(x), a)**)**

Obsevemos que *isdef*(a, x) ⇒ g(x) = a[x]. La generalización a *p* llamadas recursivas (K_1, ... ,K_p, con p>2) resulta natural. Podrían considerarse *p* maps auxiliares para simplificar la definición.

De la discusión previa surge la motivación de formular una variante de la regla PD que especifique el computo de sólo los subproblemas estrictamente necesarios, como en *memorización*, pero que conserve las ventajas de *programación dinámica*. Esta alternativa debe evidenciarse en restricciones al orden involucrado sobre el dominio del problema. El enfoque a seguir consiste en definir un orden paramétrico en el dominio, que refleje la inversión del flujo computacional sobre el árbol de cómputo de cada problema. Sea *f* una función determinística que respeta el siguiente esquema de programa (con p≥2):

funct f = (m x: P(x)) n:
  if B(x) then H(x) else E'(x, f(K_1(x)), ..., f(K_p(x))

Para cada elemento inicial ***x*** que cumpla una precondición P, el orden lineal que computa los valores desde un elemento mínimo c_x hasta el valor deseado ***x***, tendría que ser generado en base a, solamente, las restricciones subsiguientes. *z < y* significa que *z* debe computarse antes que *y*, es decir, la solución de *z* contribuye a la de *y*; ≤ es la extensión natural de <, con la igualdad.

- ¬B(**x**) |- $c_x \le (K_i(\mathbf{x})) < \mathbf{x}$, ∀i: 1≤i≤p, con p≥2
(cada subproblema "no fácil" es menor que el problema **x** a computar)

- B(z) ∧ z < y |- $c_x \le z$,
(conjuntamente con la restricción previa y las subsiguientes, c_x es un mínimo respecto a x)

- B(c_x) ≡ true
(el mínimo elemento en el orden es "fácil")

- ¬B(z) ∧ z < y |- $K_i(z) < z \wedge K_i(z) < y$, ∀i: 1≤i≤p, con p≥2
(extensión de la primera condición al árbol de cómputo del problema inicial: pseudo-transitividad)

- ∀m y: y < y ≡ false
(el orden < es irreflexivo)

[4] PD genera una recursión de cola a partir de una no lineal. Entonces, en el proceso de transformaciones que optimizan soluciones, podemos obtener directamente un loop luego de aplicar PD. No sucede así con memorización.

- $z < y \wedge z' < y \wedge z \neq z'$ |- $z < z' \veebar z' < z$
- $y < z \wedge y < z' \wedge z \neq z'$ |- $z < z' \veebar z' < z$

(pseudo-conexión respecto de los elementos del dominio intervinientes en el orden)

- $suc_x(y) \equiv (\text{that m } y' : y < y' \wedge (\forall m\ z : y < z \Rightarrow y' \leq z))$

(definición de la función sucesor para el orden < en torno a un elemento inicial *x* y un mínimo c_x)

$suc_x(x)$ puede especificarse de la siguiente manera: $suc_x(x) = \text{some m } y : [K_i(y) = x]$, para algún i entre 1 y p. Denotaremos al orden previo referido a un elemento distinguido *x*: $Ord_x\ (m_x, <_x, c_x)$. Donde $m_x = \{ y \in m \mid P(y)\ \Delta\ y \leq_x x\}$, con $<_x$ el orden < para *x* fijo de *m*, especificado precedentemente y, c_x un elemento mínimo respecto a las hojas del árbol de subproblemas necesarios para el cómputo de *x* (conjunto donde vale B). Igual que en la regla PD, *f* debe satisfacer DEF[f(suc_x(y))] |- DEF[f(y)], con $y \in m_x$. Esta condición implica, conjuntamente con el hecho P(x) ≡ true, que el orden definido está restringido sobre la precondición P y más importante aún, que la función *f* termina sobre m_x, si *f* está definida en *x* (que es asumido). Notemos que $\leq_x$ resulta un orden total sobre el conjunto m_x, con mínimo elemento c_x . Esto se sigue fundamentalmente de la pseudo-conexión y del hecho que m_x es finito (*f* termina).

El orden $<_x$ es absolutamente dependiente del problema a computar y no está definido para todo el dominio. Sobre el dominio *m* deben satisfacerse, para cada elemento *x*, las condiciones especificadas a fin de que no se computen valores innecesarios en el cálculo de *x*. Esto es, tenemos conceptualmente una familia de órdenes sobre *m*, es decir, un orden paramétrico ∠ que representa a la familia $<_x$. La función *suc* es la familia de funciones suc_x para cada *x* de *m*. Denotaremos *Ord(m, ∠)* al orden paramétrico definido.

Naturalmente las condiciones previas resultan, en general, más difícil de definirse y verificarse, respecto a las impuestas a la regla PD, en la instanciación para un problema particular, de una regla de programación dinámica que las contemple. Sin embargo son exigencias necesarias en relación a la optimización buscada.

En el ejemplo 1 (subsección 3.1) no se satisfacen las restricciones formuladas para el orden definido y de esta manera se computan valores innecesarios para determinados problemas –como analizamos al comienzo de esta subsección. En el ejemplo 2 (subsección 3.2) las condiciones formuladas se satisfacen, ya que cada elemento *x* determina un subdominio m_x –el de sus subsecuencias– sobre el cual valen las condiciones precedentes impuestas al orden para el cómputo, según la estrategia programación dinámica. suc_x es la función *suc* definida en 3.2 y c_x es la secuencia unitaria *x*[|*x*| : |*x*|]. Esto es, tanto suc_x como c_x se definen uniformemente para cada *x*. Como analizamos en 3.2.1, el orden resulta definido *exclusivamente* para todos los elementos del dominio necesarios para el cómputo de un elemento *x* dado y esto para todo *x*, es decir no se computan valores innecesarios. Luego, la solución al problema de multiplicar "con mínimo costo" *n* matrices, dada en 3.2, es una clara representante de la optimización de *programación dinámica* propuesta.

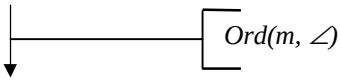
En consecuencia, bajo las restricciones formuladas al orden *Ord(m, ∠)* podemos concebir una nueva regla de *programación dinámica* que sume las ventajas de la estrategia *memorización*. Esto es, que la función resultante en el 'output scheme' no recalcule subproblemas, invirtiendo el orden computacional y eliminando de esta manera recursiones no lineales. Pero además, que no calcule subproblemas innecesarios en el cómputo de los problemas (iniciales). Seguidamente se formaliza la regla *PD&Mem* que expone el análisis desarrollado.

Ord(m, ∠) es el orden parmétrico $Ord_x\ (m_x, <_x, c_x)$, para cada *x* de *m*, que cumple las restricciones previamente formuladas. m* es el sort m restringido a $m_x \cup \{suc_x(x)\}$. c_x satisface: $c_x = \text{that m } y : [\neg\exists\ m\ z : z <_x y]$. $suc_x(y) \equiv (\text{that m } y' : y <_x y' \wedge (\forall m\ z : y <_x z \Rightarrow y' \leq_x z))$. El espacio de almacenamiento requerido sobre el Map es obviamente finito y se relaciona con el cardinal del máximo m_x de *m*. El costo de acceso al Map (inserción y recuperación) es considerado O(1).

programación dinámica + memorización

(PD&Mem)

E where
  funct f = (m x: P(x)) n:
if B(x) then H(x) else E'(x, f(K_1(x)), ..., f(K_p(x)) fi



E [g for f] where
funct g = (m x: P(x)) n:
    tabf(c_x, []) [x] where
        mode tabmap = MAP(m, n, =)
        funct tabf = (m* y, tabmap a: P(y) ∧ y ≤$_x$ suc$_x$(x)) tabmap:
            if y = suc$_x$(x) then a
            elsf B(y) then tabf (suc$_x$ (y), a[y] ← H(y))
                       else tabf (suc$_x$ (y), a[y] ← E'(x, a[K_1(x)], ..., a[K_p(x)])) fi.

Técnicas de conteo sobre m_x permiten establecer una biyección sobre un subconjunto finito de números naturales. Este isomorfismo de conteo resulta útil en la determinación del orden necesario sobre m_x, como asi también en la implementación del Map en un lenguaje que no admita tipos arbitrarios en el dominio del mismo.

5. Conclusiones y Trabajos Futuros

La regla PD introducida constituye una herramienta muy útil en la resolución de cierta clase de problemas, ya que permite transformar soluciones de manera rigurosa, asegurando en gran medida la corrección. Permite además diferenciar, en el análisis de problemas, una especificación operacional –programa recursivo ineficiente– de otra más eficiente –programa recursivo optimizado, sin recursión no lineal y sin repetición en el cómputo de subproblemas. La optimización refiere fundamentalmente al tiempo de ejecución y no al espacio de almacenmiento, ya que este último se incrementa por la introducción de un MAP en la solución.

En el ejemplo de multiplicación de matrices se puso de manifiesto la utilidad de conceptos y herramientas simples de *análisis de algoritmos*, en la deducción de un orden necesario sobre el dominio del problema, para la aplicación de la regla. Asimismo, el análisis matemático nos permitió evaluar y sacar conclusiones de la optimización respecto a la solución inicial y a la complejidad inherente al problema en sí.

Luego, para la optimización de ciertos programas recursivos ineficientes, podemos aplicar *programación dinámica*, siguiendo un proceso sistemático y formal, cuyo paso esencial consiste en la determinación de un orden lineal sobre el dominio del problema (decisiones de diseño en la metodología propuesta). Es en este paso donde usar herramientas de análisis de algoritmos puede ayudarnos a deducir el orden adecuado, sobre el conjunto de los subproblemas involucrados. Teniendo en cuenta que, debemos encontrar un elemento mínimo y un encadenamiento a partir del mismo, tal que para cada elemento *x*, si *x* depende de un elemento *y* para computarse, entonces sea $y \angle x$. Así, el problema original será el último a computar. Además, resulta importante resaltar que si por la introducción del nuevo orden de cómputo no se calculan subproblemas innecesarios, el nuevo algoritmo será una solución optimal del algoritmo original, respecto de la cantidad de subproblemas imprescindibles por resolver. No necesariamente la solución óptima respecto a cualquier algoritmo que resuelva el problema. En el ejemplo de multiplicación de matrices obtuvimos una solución optimal, aunque no siempre PD las genera.

La generalización de la regla PD hace explícitas las condiciones del orden involucrado sobre el dominio de un problema. Ésta se obtiene incorporando dos nuevas restricciones. La regla DC&PD combina la técnica de operacionalización *divide and conquer recursivo* (DC) con la de optimización *programación dinámica*. DC&PD especifica las condiciones necesarias para una aplicación conjunta de ambas estrategias, que generalmente se aplican secuencialmente en el

contexto de un desarrollo de software por transformación de programas. La regla PD&Mem suma las ventajas de la estrategia *memorización* a las de *programación dinámica*. Evita recalcular subproblemas, invirtiendo el orden computacional y eliminando recursiones no lineales. Adicionalmente impide que el programa resultante calcule subproblemas innecesarios en el cómputo, efectuando una tabulación exacta. La solución obtenida al aplicar PD&Mem será optimal respecto del número de elementos del dominio necesarios en el cómputo de cada problema. Las herramientas de análisis de algoritmos analizadas en el ejemplo de multiplicación de matrices resultan útiles, no sólo en la evaluación de los resultados, sino en la determinación del orden sobre cada subdominio m_x (finito) para el orden Ord_x $(m_x, <_x, c_x)$, que especifica la condición esencial para la aplicación de la regla PD&Mem. Un trabajo relacionado con esta última estrategia es [Boiten, 92], que define tabulaciones a partir de órdenes compatibles.

En este trabajo se analizó solamente una de las estrategias conocidas de diseño de algoritmos y de ella se dio una caracterización, con algunas extensiones. Caracterizaciones alternativas de ésta y otras técnicas (*Divide and Conquer*, *Local Search*, *Greedy, Bactracking*) pueden encontrarse en [Partsch, 90], [Besso, 94] y [Grinspan, 95a], entre otros. Futuros trabajos pueden abocarse a la modelización de técnicas aquí no tratadas, crear nuevas e incluso, dar formulaciones diferentes de la estrategia abordada. Además, estudiar los conceptos y herramientas aplicables de la rama de *análisis de algoritmos* en la derivación de las condiciones necesarias para la aplicación de las estrategias de diseño de algoritmos formalizadas (como en el ejemplo de multiplicación de matrices, para las reglas PD y PD&Mem). Asimismo, profundizar en el análisis de procedimientos que combinen las ventajas de *programación dinámica* y *memorización*. Por ejemplo, analizar bajo que condiciones es preferible en costo generar el árbol de las referencias a los subproblemas para un problema dado y a partir de él establecer el orden lineal especificado como Ord_x, para cada x. Esto es, construir dinámicamente el orden antes de computar la solución a un problema, siguiendo la metodología de *programación dinámica*.

Referencias

[Aho, 83] Aho, A. V.; Hopcroft, J. E. y Ullman, J. D., "*Data structures and algorithms*". Reading, Mass.: Addison-Wesley 1983.

[Bauer, 85] Bauer, F. L.; Berghammer, R.; Broy, M.; Dosch, W.; Geiselbrechtiger, F.; Gnatz, R.; Hangel, E.; Hesse, W.; Krieg-Brückner, B.; Lauth, A.; Matzner, T.; Möller, B.; Nickle, F.; Partsch, H.; Pepper, P.; Samelson, K.; Wirsing, M.; Wössner, H.: "The Munich proyect CIP". Volume I: The wide spectrum language CIP-S. Lecture Notes in Computer Science 183, Berlin: Springer 1985.

[Bauer 87] Bauer, F. L.; Möller, B; Partsch, H.; Pepper, P.: "The Munich proyect CIP". Volume II: The transformation system CIP-S. Lecture Notes in Computer Science 292, Berlin: Springer 1987.

[Besso, 94] Besso Pianetto, M.; Grinspan, V.; Luna, C.: "Especificación de reglas de técnicas de diseño de algoritmos". Reporte no publicado. Río Cuarto 1994.

[Bird, 80] Bird, R. S.: "Tabulation techniques for recursive programs". ACM Computing Surveys 12:2, 403-417, 1980.

[Boiten, 92] Boiten, E. A.: "Improving *recursive functions by inverting the order of evaluation*". Science of Computer Programming 18 (1992) 139-179, 1992.

[Grinspan, 95a] Grinspan, V.; Luna, C.: "Formalización de técnicas de diseño de algoritmos mediante reglas de transformación de programas". 24 JAIIO, Bs. As, Argentina 1995.

[Grinspan, 95b] Grinspan, V.; Luna, C.: "Formalización de técnicas de diseño de algoritmos mediante reglas de transformación de programas. CASOS DE ESTUDIO". CACIC'95, UNS, Bahía Blanca, Argentina, 1995.

[Manna, 74] Manna, Z: "Mathematical theory of computation". New York: McGraw-Hill 1974.

[Partsch, 90] Partsch, H. A.: "Specification and Transformation of Programs". Springer-Verlag Berlín Heidelberg , 1990.

[Sedgewick, 96] Sedgewick, R; Flajolet P.: "Analysis of Algorithms". Addison-Wesley, 1996.

[Turner, 81] Turner, D.: "The semantic elegance of applicative languages". Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, Portsmouth, N. H., 1981, pp. 83-92.