FORMALIZACIÓN DE TÉCNICAS DE DISEÑO DE ALGORITMOS MEDIANTE REGLAS DE TRANSFORMACIÓN DE PROGRAMAS. CASOS DE ESTUDIO

Valentina Graciela Grinspan* **Carlos Daniel Luna****

Departamento de Matemática. Facultad de Ciencias Exactas Físico-Químicas y Naturales.
Universidad Nacional de Río Cuarto; enlace rutas 8 y 36, km. 603 (5800) Río Cuarto.

* e-mail: vgrinspan@unrccc.edu.ar ; dir.: Avenida Italia 1207 (5800) Río Cuarto; TE: (058) - 624889
**e-mail: cluna@unrccc.edu.ar ; dir.: Ranqueles 159 (5800) Río Cuarto; TE: (058) - 631904

Palabras claves. Estrategias de diseño de algoritmos. Especificación y transformación de programas.

Resumen. A través de los años se han identificado diversas técnicas (estrategias) generales que a menudo producen algoritmos eficientes para la resolución de muchas clases de problemas. En [Grinspan, 95] dimos una caracterización y formalización de algunas de estas técnicas basándonos en el cálculo transformacional desarrollado en el proyecto CIP [Bauer, 85] [Bauer, 87]. En este trabajo utilizamos algunas de las reglas allí presentadas para resolver ciertos problemas interesantes de la programación.

Agradecemos a Gabriel Baum (LIFIA, UNLP, UNRC) por habernos guiado e insentivado permanentemente en el desarrollo de este trabajo.

FORMALIZACIÓN DE TÉCNICAS DE DISEÑO DE ALGORITMOS MEDIANTE REGLAS DE TRANSFORMACIÓN DE PROGRAMAS. CASOS DE ESTUDIO

Resumen. A través de los años se han identificado diversas técnicas (estrategias) generales que a menudo producen algoritmos eficientes para la resolución de muchas clases de problemas. En [Grinspan, 95] dimos una caracterización y formalización de algunas de estas técnicas basándonos en el cálculo transformacional desarrollado en el proyecto CIP [Bauer, 85] [Bauer, 87]. En este trabajo utilizamos algunas de las reglas allí presentadas para resolver ciertos problemas interesantes de la programación.

Palabras claves. Estrategias de diseño de algoritmos. Especificación y transformación de programas.

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. Motivados en derivar descripciones operacionales a partir de declarativas, paso a paso, nos planteamos el objetivo de definir reglas de transformación de programas que no sólo transformen especificaciones descriptivas en otras más operacionales sino también, caractericen estrategias generales de solución a problemas tales como ***Divide and Conquer*** *(Dividir y Conquistar)*, ***Búsqueda Local, Greedy*** *(Goloso)*, ***Programación Dinámica y Backtracking*** *(Búsqueda con Retroceso)* [Aho, 83].

En trabajos anteriores hemos caracterizado y formalizado estrategias de diseño de algoritmos mediante reglas utilizando el cálculo transformacional definido en el proyecto CIP [Bauer, 85] [Bauer, 87]. En [Grinspan, 95] presentamos formalizaciones para ***Divide and Conquer, Búsqueda Local, Greedy y Programación Dinámica***.

Nuestro objetivo ahora es poner de manifiesto la utilidad práctica en la resolución de problemas de algunas de las reglas allí tratadas. A tal fin, presentamos dos problemas clásicos de la programación cuya descripción formal se da en el lenguaje de especificaciones algebraicas CIP.

El lenguaje de especificaciones referido es un lenguaje de amplio espectro que contiene las construcciones clásicas de los lenguajes de programación, además de otras declarativas como la cuantificación universal y existencial de la Lógica de Primer Orden, y otras como la descripción (***that***) y la elección (***some***)[1] .

El cálculo transformacional utilizado tiene como nociones centrales las de ***esquema de programa*** y ***regla de transformación***.

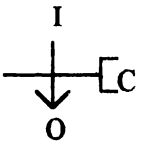
Un *programa* es un término bien formado sobre la signatura (Σ) de un tipo algebraico que define un lenguaje de programación sobre algún conjunto de tipos primitivos. Un *esquema de programa* es un término sobre Σ que contiene variables libres pertenecientes a un conjunto numerable X de parámetros tipados (en particular variables de tipo "programa"). Un esquema de programa es la generalización de un programa; los programas son esquemas de programas sin variables libres [Manna, 74].

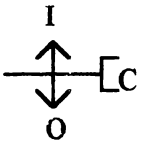
Un simple ejemplo de un esquema de programa es: E (f, x, y) que puede tener como instancias posibles, entre otras, los programas:
if $x < y$ then $f(x)$ else $f(y)$ fi , o bien
if $x < y$ then x else y fi ; donde f, x e y son los parámetros de la expresión E.

---

[1] ***some m x: P(x)*** especifica la elección de un elemento arbitrario del dominio ***m*** que satisfaga el predicado ***P***. ***that m x: P(x)*** determina al único elemento de ***m*** que satisface ***P***.

Una *regla de transformación* es una inferencia especial que denotaremos 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.

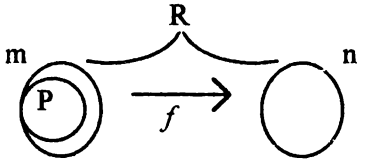
Una regla de transformación es correcta si constituye una inferencia válida, es decir, si se satisface **C** puede deducirse que **O** es un "descendiente" de **I** ($C \vdash O \subseteq I$) en el primer caso, o que **O** es "equivalente" a **I** ($C \vdash O \equiv I$) en el segundo.

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 particulares semánticos como DET y DEF [Partsch 90].

Entre los constructores descriptivos del lenguaje de especificaciones que utilizamos ($\forall$, $\exists$, some y that), el más interesante desde el punto de vista de los problemas computacionales es ***some***; ya que los problemas en general consisten en "mostrar un archivo que ....." o "encontrar un elemento tal que ....." y no solamente en responder por sí o por no ($\forall$ y $\exists$). La elección (*some*) es más general que la descripción (*that*), tiene el poder del cuantificador existencial pero además obliga a exhibir un ejemplar del dominio.

La mayor parte de este trabajo está dedicada así, a ejemplificar reglas de eliminación de ***some***, es decir, reglas de operacionalización de especificaciones descriptivas de la forma:

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



donde ***f*** es una función que retorna algún elemento de su codominio ***n*** que satisfaga el predicado ***R*** (la postcondición) para un dado elemento de su dominio ***m*** que cumpla el predicado ***P*** (la precondición).

En la sección 1 analizamos la eliminación de ***some*** por ***divide and conquer*** y en la sección 2 tratamos ***programación dinámica.*** Introducimos ambas secciones presentando la noción intuitiva de la estrategia, la justificación intuitiva de la regla y su formulación; luego nos concentramos en la aplicación práctica de cada regla introduciendo un problema cuya resolución se obtiene siguiendo los pasos:

a) matching sintáctico del 'input scheme' de la regla y decisiones de diseño,
b) instanciación y demostración de las condiciones de aplicabilidad, y
c) resultado de la aplicación de la regla.

1. DIVIDE AND CONQUER

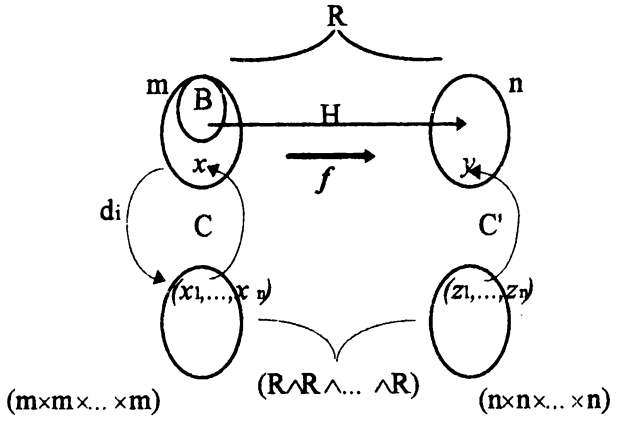
Noción intuitiva de Divide and Conquer

Esta estrategia consiste en descomponer un problema de tamaño *n* en problemas más pequeños, de modo que a partir de la solución de dichos problemas sea posible construir con "facilidad" una solución al problema completo.

Según se considere a cada subproblema del mismo tipo o no que el problema original, distinguimos entre divide and conquer recursivo y no recursivo respectivamente. Este último corresponde a la

conocida estrategia de modularización por lo que no lo trataremos, dedicándonos exclusivamente a *divide and conquer recursivo*.

Justificación intuitiva



m es el dominio del problema y ***n*** su codominio.
Cada d_i toma la componente i-ésima de tipo ***m*** de un elemento de ***m***.
C construye elementos de tipo ***m*** a partir de ***n*** elementos de dicho tipo.
C' compone las soluciones de cada subproblema obteniendo la solución de tipo ***n*** para elementos de ***m***.
R (x, y) es verdadero si y sólo si $y \in n$ es solución para $x \in m$.
B es el subconjunto de elementos de ***m*** cuya solución es considerada "fácil" y se obtiene por aplicación de la función ***H***.

Formulación de la regla: La regla propuesta en [Grinspan, 95] que formaliza la noción anterior es:

Eliminación de some por Divide and Conquer recursivo (DC)

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

1) $B(x) \equiv true \vdash R(x,H(x)) \equiv true$
2) $B(x) \equiv false \vdash [\forall\ n\ y: R(x,y) \Rightarrow R(C(d_1(x),...,d_n(x),y)] \equiv true \wedge \wedge R(d_1(x),z_1) \equiv true \wedge ... \wedge R(d_n(x),z_n) \equiv true \Rightarrow \Rightarrow R(x, C'(z_1,...,z_n)) \equiv true$
3) $B(x) \equiv false\ \Delta\ \exists\ n\ y : R(x,y) \equiv true \vdash \exists\ n\ z_1,...,z_n: R(d_1(x),z_1) \equiv true \wedge ... \wedge R(d_n(x),z_n) \equiv true$
4) $DEF[f(x)] \Leftrightarrow DEF[f'(x)]$

E [f ' for f] [2]
funct f ' = (m x : P(x)) n:
if B(x) then H(x)
else C' (f '(d_1 (x)), ..., f '(d_n(x)))

Ejemplo de aplicación: "Ordenamiento de secuencias"

Trataremos el problema de ordenar una secuencia de naturales. Este problema puede especificarse del siguiente modo[3] :

funct sort = (sequnat *x*) sequnat :
some sequnat *y*: issorted (*y*) Δ isperm (*y*,*x*); where
funct issorted = (sequnat *y*) bool : ∀ nat i : 1≤ i < |y| Δ *y*[i] ≤ *y*[i+1];
funct isperm (sequnat *y*, sequnat *x*) bool : sequtobag(*y*) = sequtobag(*x*).

donde *sequtobag* transforma una secuencia en el bag que contiene los mismos elementos.

Como la especificación del problema planteado es una instancia del 'input scheme' de la regla DC, intentaremos aplicarla para obtener una solución más operacional de nuestro problema.

---

[2] E[f' for f] denota la substitución en E de f por f '.
[3] En [Partsch, 90] se especifica este problema de la misma manera.

a) Matching sintáctico del 'input scheme' de la regla

Instancia	Expresión sintáctica del 'input scheme' de la regla DC
sequnat	m
sequnat	n
sort	f
P	true
issorted (x) $\wedge$ isperm (y,x)	R(x,y)

Decisiones de diseño

En las soluciones del problema de sorting se distinguen dos clases: *hard-split-easy-join* que reúne las soluciones en donde la descomposición del problema es "difícil" pero la composición de los resultados es sencilla y, *easy-split-hard-join* que alberga las soluciones de descomposiciones simples y composiciones complejas.

De acuerdo a las decisiones que se tomen se obtendrán soluciones pertenecientes a una u otra clase.

Clase 'hard-split-easy-join'

I) Quicksort

Decisiones de diseño

$d_1(x)$ = that sequnat s: [$\forall$ nat y : $y \in s \Leftrightarrow (y \in x \wedge y <$ findpivot(x)]

$d_2(x)$ = <findpivot (x)>

$d_3(x)$ = that sequnat s: [$\forall$ nat y : $y \in s \Leftrightarrow (y \in x$-findpivot($x$) $\wedge$ y $\geq$ findpivot (x)]

funct findpivot = (sequnat s: $s \neq <>$) nat : that nat z : $z \in s \wedge$ cond (s, z)

(cond(s, z) es verdadero sii z satisface el criterio de selección del pivote que dejamos a implementación)

$B(x) \equiv (|x| \leq 1)$ $H(x) = x$

$C' = +$ $C = +$ (+ denota la concatenación de secuencias)

b) Analicemos ahora si las condiciones de aplicabilidad se satisfacen según las instancias anteriores:

1) $(|x| \leq 1) \equiv$ true $\vdash$ (issorted (x) $\wedge$ isperm (x,x)) $\equiv$ true. Se cumple trivialmente.

2) $(|x| > 1) \equiv$ true $\vdash$ [$\forall$ n y R(x,y) $\Rightarrow$R(C($d_1(x)$,$d_2(x)$,$d_3(x)$),y)] $\equiv$ true $\wedge$
$\wedge$ [issorted(z_1) $\wedge$ isperm(z_1,$d_1(x)$)] $\equiv$ true $\wedge$ [issorted(z_2) $\wedge$ isperm(z_2,$d_2(x)$)] $\equiv$ true $\wedge$
$\wedge$ [issorted(z_3) $\wedge$ isperm(z_3,$d_3(x)$)] $\equiv$ true $\Rightarrow$ [issorted($z_1+z_2+z_3$) $\wedge$ isperm($z_1+z_2+z_3$, x)] $\equiv$ true

Sea x una secuencia de más de un elemento tal que toda solución y de x es solución de C($d_1(x)$,$d_2(x)$,$d_3(x)$) y además si z_1 , z_2 y z_3 son las permutaciones ordenadas de $d_1(x)$, $d_2(x)$ y $d_3(x)$ respectivamente, entonces puede verse que, de acuerdo a las definiciones de d_1, d_2 y d_3, ($z_1 + z_2 + z_3$) es la permutación ordenada de x.

3) $(|x| > 1) \equiv$ true Δ $\exists$ sequnat y : [issorted (x) $\wedge$ isperm (y,x)] $\equiv$ true $\vdash$ $\exists$ sequnat z_1 , z_2 , z_3 :
[issorted (z_1) $\wedge$ isperm (z_1,$d_1(x)$)] $\equiv$ true $\wedge$ [issorted (z_2) $\wedge$ isperm (z_2,$d_2(x)$)] $\equiv$ true $\wedge$
$\wedge$ [issorted (z_3) $\wedge$ isperm (z_3,$d_3(x)$)] $\equiv$ true

Debemos demostrar ahora que si x es una secuencia de más de un elemento y existe su permutación ordenada, entonces existen también los ordenamientos para $d_1(x)$, $d_2(x)$ y $d_3(x)$ respectivamente.

Podemos probar esto estableciendo que para cualquier secuencia de elementos sobre los que esté definido un orden total, existe una permutación ordenada. Demostremos entonces esto último.

Demostración por inducción sobre la cantidad de elementos de la secuencia:

Sea s una secuencia de elementos sobre los que está definida una relación de orden total.

- Caso base: $|s| \leq 1$. Obviamente s es la permutación ordenada de sí misma.

- Supongamos que las secuencias de longitud n poseen la propiedad a establecer.
- Sea $|s| = n+1$, demostremos la propiedad para s. Existe en s un elemento mínimo e, tal que $e + (s - e)$ (donde .+. es la operación de adicionar un elemento a la cabeza de una secuencia y .-. la función que suprime un elemento de una secuencia) es una permutación de s. Pero por la hipótesis inductiva $s - e$ admite una permutación ordenada s', luego $e + s'$ es la permutación ordenada de s que buscábamos.

4) Como la función sort está definida sobre el conjunto de las secuencias de naturales sin restricción al igual que la función quicksort; es válida la condición: DEF [sort(x)] $\Leftrightarrow$ DEF[quicksort(x)]

c) Por los incisos a) y b) precedentes, podemos aplicar la regla DC obteniendo una descripción más operacional de *sort* que corresponde a una especificación declarativa del conocido algoritmo de ordenación *quicksort*. Llamando quicksort a sort ' (f' en DC):

funct quicksort (sequnat s : true) sequnat:
if $|s| \leq 1$ then s else quicksort ($d_1(s)$) + quicksort ($d_2(s)$)* + quicksort ($d_3(s)$) fi.

* con quicksort ($d_2(s)$) = $d_2(s)$.

II) Selectsort

El algoritmo de ordenación conocido como *selectsort* es una instancia de quicksort en donde la condición de selección del pivote *cond(s, z)* es "z es *el mínimo elemento de la secuencia s*" y por lo tanto, la partición que contiene a todos los elementos menores que el pivote es vacía. De esta manera reproducimos el ejemplo anterior con los siguientes cambios:

$d_2(x) = <$ nat z: $z \in x \wedge \forall$ nat y: $(y \in x \Rightarrow z \leq y)> = <$ findpivot (x)>
$d_3(x) =$ (x - findpivot (x))

y llamando selectsort a sort ' obtenemos:

funct selectsort (sequnat s : true) sequnat:
if $|s| \leq 1$ then s else selectsort ($d_2(s)$)* + selectsort ($d_3(s)$) fi.

* con selectsort (d_2(s)) = d_2(s)

Clase 'easy-split-hard-join'

III) Mergesort

Decisiones de diseño:
$d_1(x)$ = that sequnat s_1 : $s_1 \neq <> \wedge$ ($\exists$ sequnat s_2 : $s_2 \neq <> \wedge s_1 + s_2 = x$) $\wedge$ $|s_1| = k(x)$
(k especifica la longitud de la primera particición de x; su definición queda a implementación).
$d_2(x)$ = that sequnat s_2 : $d_1(x) + s_2 = x$
C = +
(funct C ' = (sequnat z_1, z_2 : $z_1 \neq <> \wedge$ $z_2 \neq <> \wedge$ issorted(z_1) $\wedge$ issorted(z_2)) sequnat :
some sequnat s: issorted (s) $\wedge$ isperm (s, z_1+z_2)) =def. merge (z_1,z_2)
B(x) $\equiv$ ($|x| \leq 1$) H(x) = x

b) Analicemos ahora si las condiciones de aplicabilidad se satisfacen teniendo en cuenta las nuevas decisiones de diseño y recordando el matching sintático del 'input scheme' (tabla del inciso a)).

1) ($|x| \leq 1$) $\equiv$ true $\vdash$ (issorted (x) $\wedge$ isperm (x,x)) $\equiv$ true. Se cumple trivialmente.

2) ($|x| > 1$) $\equiv$ true $\vdash$ [$\forall$ n y: R(x,y) $\Rightarrow$R(C($d_1(x)$,$d_2(x)$),y)] $\equiv$ true $\wedge$
$\wedge$ [issorted(z_1) $\wedge$ isperm(z_1 ,$d_1(x)$)] $\equiv$ true $\wedge$ [issorted(z_2) $\wedge$ isperm(z_2,$d_2(x)$)] $\equiv$ true $\Rightarrow$
$\Rightarrow$ [issorted(merge(z_1 ,z_2)) $\wedge$ isperm(merge(z_1 ,z_2), x)] $\equiv$ true

Sea x una secuencia de más de un elemento tal que toda solución de x es solución de C($d_1(x)$,$d_2(x)$);

además sean z_1 y z_2 las permutaciones ordenadas de $d_1(x)$ y $d_2(x)$ respectivamente, entonces por la definición de merge el predicado [issorted (merge(z_1 , z_2)) $\wedge$ isperm (merge(z_1 , z_2), x)] resulta verdadero.

3) ($|x| > 1$) $\equiv$ true Δ $\exists$ sequnat y : [issorted (x) $\wedge$ isperm (y,x)] $\equiv$ true $\vdash$ $\exists$ sequnat z_1, z_2 :
[issorted (z_1) $\wedge$ isperm(z_1,$d_1(x)$)] $\equiv$ true $\wedge$ [issorted (z_2) $\wedge$ isperm(z_2,$d_2(x)$)] $\equiv$ true

La tesis vale con independencia de la hipótesis pues ya demostramos en quicksort que para toda secuencia de naturales existe un ordenamiento.

4) Como la función sort está definida sobre el conjunto de las secuencias de naturales sin restricción al igual que mergesort, es válida la condición : DEF [sort(x)] $\Leftrightarrow$ DEF[mergesort(x)]

c) Como las condiciones de aplicabilidad se cumplen podemos aplicar la regla, obteniendo una descripción más operacional de *sort* que corresponde a una especificación declarativa del algoritmo de ordenación *mergesort*. Llamando mergesort a sort ':

funct mergesort (sequnat s : true) sequnat:
if $|s| \leq 1$ then s else merge (mergesort ($d_1(s)$), mergesort ($d_2(s)$)) fi.

IV) Insertsort

Insertsort es un caso particular de mergesort en donde la función k que determina la primera partición de la secuencia de entrada x es $k(x) = 1$. Mientras que mergesort divide x en dos secuencias cuyo tamaño no es explicitado, insersort la particiona en cabeza y cola. De esta manera las particiones de x resultan:

$d_1(x) = <$ first (x)>
$d_2(x) =$ rest (x)

y llamando insertsort a sort' obtenemos una especificación descriptiva del algoritmo de ordenamiento conocido como *insertsort*:

funct insertsort (sequnat s : true) sequnat:
if $|s| \leq 1$ then s else merge (insertsort($d_1(s)$)*, insertsort ($d_2(s)$)) fi.

* donde insertsort(d_1 (s)) = d_1 (s)

2. PROGRAMACIÓN DINÁMICA

Noción intuitiva

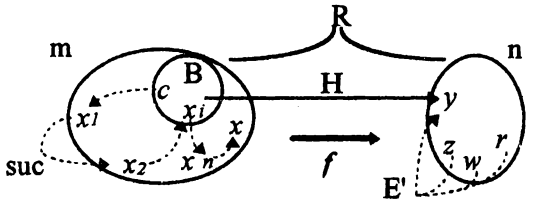
A menudo la solución a un problema se obtiene dividiendo el problema en tantos subproblemas como sea necesario, y luego dividiendo cada subproblema en subproblemas más pequeños produciendo un algoritmo de tiempo exponencial. No obstante, con frecuencia, sólo hay un número polinomial de subproblemas, 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.

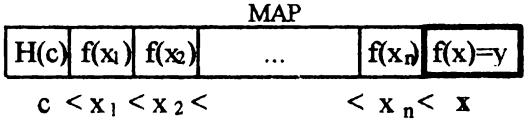
Desde el punto de vista de la realización, algunas veces es más fácil crear una tabla de las soluciones de todos los subproblemas que se tengan que resolver. Se rellena la tabla sin tener en cuenta si se necesita realmente un subproblema particular en la solución total. La formación de la tabla de subproblemas para alcanzar una solución a un problema dado se denomina programación dinámica.

Aspectos relevantes : La forma de un algoritmo de programación dinámica puede variar, pero hay un esquema común : una tabla a llenar (que implementamos sobre un MAP) y un orden en el cual se hacen las entradas, que corresponde a un orden total 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.

Justificación intuitiva





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

m es el dominio del problema 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 sii ***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.

Formulación de la regla: La regla presentada en [Partsch, 90] que formaliza las nociones previas es

Programación Dinámica
(PD)

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

1) LNSORD (m,<,c)
2) suc (x) ≡ (that m y : x < y ∧ (∀m z: x < z ⇒ y ≤ z))
3) 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.
```

Ejemplo de aplicación: "Multiplicación de matrices"

Se tienen n matrices M_1 , M_2 , ... , M_n de índices naturales y elementos de tipo r. Consideramos definidas las siguientes funciones:
filas (Mi) = "número de filas de la matriz Mi "
columnas (Mi) = "número de columnas de la matriz Mi "

Llamaremos $M_{i..j}$ a la matriz resultado del producto M_i * ... * M_j , con $i \leq j$. 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 calcular $M_{i..j}$ (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 calcular $M_{i..j}$ se puede elegir la asociación que involucre menos operaciones. Por ejemplo, sean las matrices M_1 de dimensión 10×20, M_2 de 20×30 y M_3 de 30×30; resulta:
costo $((M_1 * M_2) * M_3) = 10*20*30 + 10*30*30 = 15000$, mientras que
costo $(M_1 * (M_2 * M_3)) = 20*30*30 + 10*20*30 = 24000$.

Se desea especificar un algoritmo para calcular el mínimo costo de obtener $M_{1..n}$.
A continuación damos la formalización de una primera solución al problema,

```
mode[4] 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 elementos de tipo matriz}

funct f = (sequmatriz z : z ≠ <> ∧ ∀ 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
```

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 (∞).

Notamos $x[i : j]$ a la subsecuencia de x entre los índices i y j, con $1 \le i \le j \le |x|$.

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 para una entrada concreta para notar que algunos subproblemas se resuelven muchas veces.

Para optimizar el proceso de cómputo de f intentaremos aplicar la regla PD y obtener un 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 de sequmatriz de todas sus subsecuencias y definiremos sobre m un orden total estricto con mínimo elemento, como veremos luego.

Cada entrada inicial x de f determina unívocamente su subconjunto m, y además como f restringida a m (f /m) es una función bien definida para todas sus entradas, podemos aplicar la regla PD a f /m.

---

[4] *mode* es una notación que permite abreviar instancias de tipos.

a) 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 : $z \subseteq x \wedge$ $\wedge \forall$ nat i : $1 \leq i < \|x\| \Rightarrow$columnas (x[i]) = filas (x[i+1]))	m (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 ≤ j then i else j fi	E'(f, x)

Decisiones de diseño:

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 $x[i : j]$) m : [5]

if ($i = 1 \wedge j = |x|$) then <>
elsf ($j = |x|$) then $x[i-1: i-1]$
else $x[i : j+1]$ fi.

Obviamente c = $x[|x| : |x|]$ es el mínimo elemento. También en este caso existe un último elemento, aunque la regla no lo exige, que es suc($x[1: |x|]$) = <>.

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

1) LNSORD (m, <, $x[|x| : |x|]$)

Es fácil demostrar que la relación $\angle$ define un orden total estricto sobre m con mínimo elemento $x[|x| : |x|]$.

2) suc(z) ≡ (that m y : $z \angle y \wedge \forall$ m w: ($z \angle w \Rightarrow y \trianglelefteq w$))

Sabemos por definición de la relación $\angle$ que $z \angle$ suc(z) , con lo cual probamos que suc(z) satisface el primer conyungendo de esta condición. Probemos ahora que satisface el segundo :

Sea w un elemento de m tal que $z \angle w$ entonces, por definición de $\angle$ suc(z) = $w \vee$ suc(z) $\angle w$, y esto equivale a suc(z) $\trianglelefteq w$.

Puesto que suc(z) es único, está definido como lo requiere la condición 2).

---

[5] Formalmente el perfil de suc debería ser suc (m x , nat i, j) m , pero usamos la notación x[i : j] por simplicidad.

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

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

```
funct f' = (m x) nat :
   tabf (x[lxl : lxl] ) [ x[1 : lxl] ] where

mode tabmap = MAP ( m, nat, = );
funct tabf  = ( m y, tabmap a :  y ⊴suc(x) ∧ ∀ m 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 = (m 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 z subsecuencia de x vale:

$$f(z[1: k]) = a[z[1: k]] \quad \text{y} \quad f(z[k+1: |z|]) = a[z[k+1: |z|]]$$

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:

(a) $x[|x| : |x|] \trianglelefteq z1 \angle y$

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

(b) $x[|x| : |x|] \trianglelefteq z2 \angle y$

Nuevamente, la primera desigualdad se establece en virtud de la minimidad de $x[|x| : |x|]$. La segunda puede deducirse de la definición de $\angle$ ya que, según la definición de suc,

$$suc^{a}(z2) = y \quad \text{con} \quad a = ((k-i+1)*(|x|+1)) - i - (k*(k+1) - i*(i+1)) / 2 .$$

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 respectivamente 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 orden : $O(n^3)$).

CONCLUSIONES

Este trabajo pretende contribuir a resolver uno de los problemas centrales de la programación : *cómo se construyen soluciones, descriptas por programas, para resolver problemas.*

Las reglas introducidas en [Grinspan, 95] constituyen una herramienta muy útil en la resolución de ciertas clases de problemas ya que permiten construir soluciones de manera sistemática y formal, asegurando en gran medida su corrección. Permiten además diferenciar, en el análisis de problemas, la primera especificación descriptiva de su resolución (especificación más operacional).

En nuestro trabajo utilizamos como mecanismo de resolución de problemas reglas que modelizan una posible caracterización de las estrategias 'divide and conquer' y 'programación dinámica'. Otras caracterizaciones de estas y otras estrategias pueden encontrarse en [Besso Pianetto, 94] y [Grinspan, 95].

Futuros trabajos pueden abocarse a la modelización y ejemplificación de técnicas aquí no tratadas, crear nuevas e incluso, dar caracterizaciones diferentes de las estrategias abordadas. Sería interesante también estudiar el poder expresivo de los distintos formalismos planteados.

Para concluir, dejamos la siguiente reflexión: *analizar problemas no es trivial, y ... ¿ programar?.*

REFERENCIAS BIBLIOGRÁFICAS

[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.

[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.

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

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