

**UNED. Centro Asociado de VITORIA-GASTEIZ****Escuela Técnica de Informática****Programación II****Ejercicio ejemplo 2****Curso 1997/98**

Los presentes apuntes han sido elaborados para su distribución gratuita a los alumnos de Programación II de Informática de Sistemas y Gestión del CENTRO ASOCIADO DE VITORIA-GASTEIZ de la Escuela Técnica de Informática de U.N.E.D.. Se ofrecen para uso personal de dichos alumnos pudiéndose reproducir y transmitir siempre que se mantenga esta nota y no se realice actividad de intercambio comercial de ningún tipo. En cualesquiera otras condiciones la copia, transmisión o almacenamiento por cualquier medio fotográfico, informático, telemático u otros requiere la autorización expresa del autor. Se agradecerá la comunicación de cualquier errata así como cualquier comentario o sugerencia.

(e-mail :jqc@jet.es)

© Jerónimo Quesada 1997

**1. INTRODUCCIÓN**

En estos apuntes se desarrolla el ejercicio de diseño y verificación de algoritmos propuesto como práctica para el curso 96/97. Se trata de un ejercicio que permite recorrer las técnicas de análisis y diseño recursivo e iterativo y la obtención de distintas soluciones. En el desarrollo del ejercicio se sigue el Cuadernillo del Alumno publicado en el Departamento de la Sede Central para la práctica de Programación II del curso 96/97, se aconseja leer estos apuntes siguiendo a la vez ese Cuadernillo. Se hará referencia continua al texto base de la asignatura: "Diseño de Programas", R.Peña Marí, Ed. Prentice-Hall y se utilizará la notación definida en éste texto. Se sigue el planteamiento establecido en el citado Cuadernillo del alumno y que consiste en:

- a) Especificar rigurosamente el algoritmo
- b) Diseño de una solución recursiva no final. Se realizarán distintas aproximaciones
- c) Se obtendrá la solución recursiva final a partir de la no final transformándola por plegado-desplegado
- d) Se obtendrá la solución iterativa por transformación de la recursiva

Obsérvese que en esta descripción de objetivos siempre se habla de “**una solución**”, nunca de “**la solución**”. En las tutorías es normal escuchar frases como : “¿Cuál es la solución a este problema ?”. En toda obra de ingeniería no trivial, y el diseño de algoritmos lo es, no suele existir una solución única. Es normal que exista un conjunto de soluciones que puede ser vacío, finito o infinito. Las soluciones pueden, desde luego, dividirse entre correctas e incorrectas y un objetivo básico de la asignatura es aprender a diseñar soluciones correctas. Dentro de las soluciones correctas puede haber soluciones más o menos elegantes, eficientes, rebuscadas, interesantes, originales, ... , el juicio sobre cuál es la mejor entre dos soluciones correctas es más subjetivo que el juicio entre cuál es o no correcta y queda a discreción de los lectores (en la vida real la mejor solución es la que más satisface al cliente). No se repiten aclaraciones y desarrollos ya expuestos en el Ejercicio 1 de esta misma colección que se aconseja repasar antes de abordar éste.

## 2. ENUNCIADO

Se desea un algoritmo, completamente verificado, que evalúe un polinomio de coeficientes enteros (estos coeficientes se hallan almacenados en un vector **p:vector[1..n] de ent**, de manera que la posición **i-ésima** contiene el coeficiente de  $x^{i-1}$ ) en un punto **x** (real).

## 3. ESPECIFICACIÓN

Como siempre que se utilice un tipo de datos distinto de los predeterminados: natural, entero, booleano, real ... se ha de definir exactamente el nuevo tipo de datos. En este caso el tipo **vect** es:

tipo **vect**= vector [1..n] de enteros.

La definición del algoritmo en la notación de Peña puede ser:

**fun** poli(p: vect, x: real) **dev** y: real

En la precondition se han de establecer las condiciones que han de cumplir los parámetros de entrada. En este caso el parámetro **p** ha de ser del tipo definido **vect** , **x** un número real al que no se han de imponer, en principio, condiciones adicionales. La postcondición ha de indicar que en **y** se devuelve el resultado de evaluar el polinomio para el valor **x**., es decir la suma de los productos de cada uno de los elementos del vector por la potencia de **x** asociada. La especificación completa queda entonces:

{  $Q \equiv \text{cierto}$  }

**fun** poli(p: vect, x: real) **dev** y: real

{  $R \equiv y = \sum_{a=1}^n p[\mathbf{a}].x^{a-1}$  }

En las tutorías se ha planteado la cuestión: ¿qué ocurre cuando x vale cero y el exponente de x vale 0 ?. La función exponencial real no está definida en este caso. Pero en este ejercicio es más lógico considerar la exponencial entera definida como:

$$x^i = \begin{cases} \prod_{a=1}^i x & \text{si } i > 0 \\ 1 & \text{si } i = 0 \end{cases}$$

(véase Peña Cap 2, Convenio sobre cuantificadores y sustituciones, y en el Cap 3 el ejemplo, potencia entera, utilizado en los apartados 3.1 y 3.3). De esta forma queda definido el caso de exponente cero para  $x=0$ .

## 4. DISEÑO RECURSIVO POR INMERSIÓN NO FINAL

### 4.1. Primer diseño

Como siempre, se parte de la postcondición del algoritmo original en la búsqueda de una inmersión apropiada. Distintas inmersiones son posibles y se adelanta que se optará finalmente por una no trivial que permite obtener una solución eficiente y elegante y lo suficientemente compleja como para que sea un buen ejemplo de aplicación de las técnicas de plegado-desplegado en un apartado posterior. En este apartado se opta por la más inmediata. La postcondición original era:

$$R \equiv y = \sum_{a=1}^n p[\mathbf{a}].x^{a-1}$$

Para generalizarla la técnica básica consiste en introducir nuevas variables en la postcondición, normalmente en sustitución de constantes. En este caso se puede introducir una variable  $i$  en sustitución de la constante  $n$  y se tiene:

$$R \equiv y = \sum_{a=1}^i p[\mathbf{a}].x^{a-1} \wedge i = n$$

La postcondición original  $R$  se puede expresar entonces como:

$$R \equiv R_1 \wedge R_2 \text{ siendo } R_1 \equiv y = \sum_{a=1}^i p[\mathbf{a}].x^{a-1} \text{ y } R_2 \equiv i = n$$

La nueva postcondición es  $R_1$ , en la nueva precondition se han de incluir las condiciones que ha de cumplir la variable incorporada. Se sigue la regla de admitir el valor cero para la variable de inmersión  $i$  (véase Ejercicio 1 de esta colección). La nueva función, en la que se esboza la solución es:

$$\{ Q \equiv 0 \leq i \leq n \}$$

**fun** polil(p: vect ; x: real; i:natural) **dev** y: real

caso  $i=0 \rightarrow 0$

$\dot{y}$   $i>0 \rightarrow ?$

f caso

ffun

$$\{ R \equiv y = \sum_{a=1}^i p[\mathbf{a}].x^{a-1} \}$$

En el caso no trivial se ha de realizar una llamada recursiva a la función reduciendo el tamaño del problema, es decir, acercándose al caso trivial. Esto se consigue con:

```
{ Q1 ≡ 0 ≤ i ≤ n }
fun poli1(p: vect ; x: real; i:natural) dev y: real
  caso i=0 → 0
  ÿ i>0 → sea y' = poli(p,x,i-1)
           en ?
  fcaso
ffun
```

$$\{ R \equiv y = \sum_{a=1}^i p[\mathbf{a}].x^{a-1} \}$$

? ahora es la operación adicional que se ha de realizar tras la llamada recursiva. Tras la llamada recursiva se obtiene un valor y' que cumplirá postcondición para p e i-1, es decir:

$$y' = \sum_{a=1}^{i-1} p[\mathbf{a}].x^{a-1}$$

A partir de y' se ha de devolver un valor y que cumpla:

$$y = \sum_{a=1}^i p[\mathbf{a}].x^{a-1}$$

Es inmediato que ha de ser  $y = y' + p[i].x^{i-1}$  luego el diseño queda:

```
{ Q ≡ 0 ≤ i ≤ n }
fun poli1(p: vect ; x: real; i:natural) dev y: real
  caso i=0 → 0
  ÿ i>0 → sea y' = poli1(p,x,i-1)
           en y' + p[i].xi-1
  fcaso
ffun
```

$$\{ R \equiv y = \sum_{a=1}^i p[\mathbf{a}].x^{a-1} \}$$

que puede expresarse simplemente como:

```
{ Q1 ≡ 0 ≤ i ≤ n }
fun poli1(p: vect ; x: real; i:natural) dev y: real
  caso i=0 → 0
  ÿ i>0 → poli(p,x,i-1) + p[i].xi-1
  fcaso
ffun
```

$$\{ R \equiv y = \sum_{a=1}^i p[\mathbf{a}].x^{a-1} \}$$

Para obtener el mismo resultado con **poli1** que el pretendido originalmente del algoritmo **poli** basta con ejecutar **poli1** para  $i=n$ , es decir:  $\text{poli}(p,x)=\text{poli1}(p,x,n)$ . Se deja como ejercicio para el lector comprobar el funcionamiento de esta solución y verificarlo (puede seguirse como guía el Ejercicio 1 de esta misma colección ).

#### 4.2. Inmersión de eficiencia

En la solución presentada en el apartado anterior tras cada llamada recursiva se ha de calcular la potencia de  $x$ . Si la exponencial se puede considerar como una operación de coste constante en el sistema en que finalmente se programe el algoritmo, no hay nada que objetar. En otro caso el coste de la operación adicional no será constante, ya que supone realizar un número de multiplicaciones dependiente de  $i$ . Aplicando la recurrencia 1.2 de Peña, queda:

$$T(n) = \begin{cases} c & \text{si } n < 1 \\ T(n-1) + cn & \text{si } n \geq 1 \end{cases}$$

se tienen para los parámetros  $a, b$  y  $k$  de dicha ecuación los valores 1, 1 y 1 respectivamente, por lo que la solución es (aplicando 1.3 de Peña):

$$T(n) \in \mathcal{O}(n^2)$$

Puede mejorarse la eficiencia del algoritmo realizando una inmersión de eficiencia, si se utilizar una nueva variable de inmersión  $z$  en la que se mantiene el valor de  $x^{i-1}$  se llega a la solución:

$$\{ Q_1 \equiv 0 \leq i \leq N \wedge z = x^{i-1} \wedge x \neq 0 \}$$

**fun** poli2(p: vect ; x: real; i:natural; z: real) **dev** y: real

caso  $i=0 \rightarrow 0$

$\dot{y}$   $i>0 \rightarrow \text{poli2}(p,x,i-1,z/x)+p[i].z$

fcaso

ffun

$$\{ R \equiv y = \sum_{a=1}^i p[\mathbf{a}].x^{a-1} \}$$

Esta solución no resulta muy apropiada ya que ahora sí que exige excluir (o contemplar como caso especial) el valor  $x=0$  para evitar una división por cero, además la actualización de  $z$  exige realizar una división real y la llamada inicial es  $\text{poli2}(p,x,n,x^{n-1})$  lo que supone que se ha de calcular la potencia  $n-1$  de  $x$  antes de ejecutar la llamada.

Si en la postcondición original se realiza la sustitución:

$$R \equiv y = \sum_{a=i+1}^N p[\mathbf{a}].x^{a-1} \wedge i = 0$$

Se obtiene una inmersión distinta que da lugar al algoritmo:

```
{ Q ≡ 0 ≤ i ≤ n }
fun poli3(p: vect ; x: real; i:natural) dev y: real
    caso i=n → 0
    ÿ i<n → poli3(p,x,i+1)+p[i+1]. xi
    fcaso
ffun
    N
{ R ≡ y = ∑a=i+1N p[a]. xa-1 }
```

de llamada inicial poli3(p,x,0)

En este algoritmo es posible realizar una inmersión de eficiencia que conduce al resultado:

```
{ Q ≡ 0 ≤ i ≤ n ∧ z = xi }
fun poli4(p: vect ; x: real; i:natural; z: real) dev y: real
    caso i=n → 0
    ÿ i<n → poli4(p,x,i+1,z*x)+p[i+1]. z
    fcaso
ffun
{ R ≡ y = ∑a=i+1n p[a]. xa-1 }
```

con llamada inicial poli4(p,x,0,1). Se aconseja realizar el desarrollo completo de esta solución así como su verificación. Ésta es una solución adecuada al problema, sin embargo en el siguiente apartado se realizará una inmersión que da lugar de forma directa a una solución eficiente sin necesidad de realizar una posterior inmersión de eficiencia.

### 4.3. Inmersión eficiente

En la evaluación de un polinomio en un punto dado puede seguirse una técnica que permite no tener que recalcular la potencia  $i$ -ésima de  $x$  para cada término y que consiste ,para el ejemplo concreto de  $n=4$ , en hacer:

$$p[1]+p[2].x+p[3].x^2+p[4].x^3 = p[1]+x. ( p[2]+x. ( p[3]+x . p[4] ) )$$

A partir de esta idea se desarrolla una solución recursiva no final para el problema propuesto.

La postcondición original puede escribirse como:

$$R \equiv y = \sum_{a=i+1}^n p[\mathbf{a}].x^{a-(i+1)} \wedge i = 0$$

Obsérvese que se ha sustituido el valor 1 de la expresión original por la expresión  $i+1$  tanto en el origen del sumatorio como en el exponente de  $x$ . Para  $i=0$  ambas expresiones coinciden. Se utilizará la expresión

$$y = \sum_{a=i+1}^n p[\mathbf{a}].x^{a-(i+1)}$$

Como nueva postcondición. El caso trivial se tiene para  $i=n$ , entonces el sumatorio se anula y basta devolver el valor 0 como solución. En el caso no trivial se deberá realizar una llamada que aproxime el problema al caso trivial, es decir, una llamada en la que se incremente  $x$ . Un primer esbozo del algoritmo es:

```
{ Q ≡ 0 ≤ i ≤ n }
fun poli5(p: vect ; x: real; i:natural) dev y: real
  caso i=n → 0
  ÿ i<0 → sea y' = poli5(p,x,i+1)
           en ?
  fcaso
ffun
```

$$\{ R_1 \equiv y = \sum_{a=i+1}^n p[\mathbf{a}].x^{a-(i+1)} \}$$

? ahora es la operación adicional que se ha de realizar tras la llamada recursiva. Tras la llamada recursiva se obtiene un valor  $y'$  que cumplirá postcondición para  $i+1$ , es decir:

$$y' = \sum_{a=i+2}^n p[\mathbf{a}].x^{a-(i+2)}$$

A partir de  $y'$  se ha de devolver un valor  $y$  que cumpla:

$$y = \sum_{a=i+1}^n p[\mathbf{a}].x^{a-(i+1)}$$

Se tiene:

$$y = \sum_{a=i+1}^n p[\mathbf{a}].x^{a-(i+1)} = p[i+1].x^{i+1-(i+1)} + \sum_{a=i+2}^n p[\mathbf{a}].x^{a-(i+1)} = p[i+1] + x \cdot \sum_{a=i+2}^n p[\mathbf{a}].x^{a-(i+2)} = p[i+1] + x \cdot y'$$

De esta igualdad se deduce que  $y$  se puede obtener a partir de la suma de  $p[i+1]$  y el producto de  $x$  por  $y'$ . Se obtiene:

```

{ Q ≡ 0 ≤ i ≤ n }
fun poli5(p: vect ; x: real; i:natural) dev y: real
  caso i=n → 0
  ÿ i<n → sea y' = poli5(p,x,i+1)
           en p[i+1]+x.y'
  fcaso
ffun
{ R1 ≡ y = ∑a=i+1n p[a].xa-(i+1) }

```

Que puede expresarse como:

```

{ Q1 ≡ 0 ≤ i ≤ n }
fun poli5(p: vect ; x: real; i:natural) dev y: real
  caso i=n → 0
  ÿ i<n → poli5(p,x,i+1).x +p[i+1]
ffun
{ R1 ≡ y = ∑a=i+1n p[a].xa-(i+1) }

```

La operación adicional no depende de n, se reduce a un producto y una suma. Es, por tanto, de coste constante y no provoca un incremento del coste del algoritmo.

Obsérvese cómo funciona el algoritmo para el caso concreto n=4:

La llamada inicial es poli5(p,x,0)

Puesto que 0<4 se produce una nueva llamada: poli5(p,x,1)

Puesto que 1<4 se produce una nueva llamada: poli5(p,x,2)

Puesto que 2<4 se produce una nueva llamada: poli5(p,x,3)

Puesto que 3<4 se produce una nueva llamada: poli5(p,x,4)

puesto que 4=4 se retorna el valor 0 de la llamada poli5(p,x,4)

La llamada poli5(p,x,3) retorna el valor: 0.x+p[4]=p[4]

La llamada poli5(p,x,2) retorna el valor: p[4].x+p[3]

La llamada poli5(p,x,1) retorna el valor. (p[4].x+p[3]).x +p[2]=p[4].x<sup>2</sup>+p[3].x+p[2]

La llamada poli5(p,x,0) retorna finalmente el valor:

(p[4].x<sup>2</sup>+p[3].x+p[2]).x+p[1]= p[4].x<sup>3</sup>+p[3].x<sup>2</sup>+p[2].x+p[1] que es la evaluación del polinomio en el punto x.

#### 4.4. Verificación

Siguiendo, como siempre, la Tabla 3.2 de Peña se tiene:

1. **Completitud de la alternativa:**  $Q(\bar{x}) \Rightarrow B_t(\bar{x}) \vee B_m(\bar{x})$ . Se ha de comprobar que todos los parámetros que cumplen la precondition son contemplados en alguno de los casos triviales o no triviales. En este caso la concreción de Q, B<sub>t</sub> y B<sub>m</sub> es  $Q \equiv 0 \leq i \leq n$   $B_t \equiv i = n$   $B_m \equiv i < n$  y es inmediato que  $0 \leq i \leq n \Rightarrow i = n \vee i < n$   
En palabras: para todos los valores de i admitidos por precondition existe un caso que los trata

2. **Satisfacción de la precondition para la llamada interna:**  $Q(\bar{x}) \wedge B_m(\bar{x}) \Rightarrow Q(s(\bar{x}))$ . En otras palabras se ha de comprobar que la llamada recursiva se realiza con parámetros que cumplen precondition. En el ejemplo tratado la función  $s$  que realiza la transformación de parámetros antes de la llamada recursiva es tal que transforma la tripla  $\mathbf{p}, \mathbf{x}, \mathbf{i}$  en la  $\mathbf{p}, \mathbf{x}, \mathbf{i}+1$ , es decir  $s(\mathbf{p}, \mathbf{x}, \mathbf{i}) = \mathbf{p}, \mathbf{x}, \mathbf{i}+1$ . Sólo es necesario tener en cuenta  $\mathbf{i}$  ya que es la única variable contemplada en la precondition y en la condición de caso no trivial. La precondition aplicada a  $\mathbf{i}$  es  $Q \equiv 0 \leq i \leq N$ , aplicada a  $\mathbf{i}+1$  es  $Q_i^{i+1} \equiv 0 \leq i+1 \leq n$  y como  $B_m \equiv i < n$  se tiene:  $0 \leq i \leq n \wedge i < n \equiv 0 \leq i < n \Rightarrow 0 \leq i+1 \leq n$ . En palabras: los parámetros que se utilizan en la llamada interna cumplen la precondition general de la función. Obsérvese que esto es debido a que se garantiza que la llamada interna sólo se realiza si  $\mathbf{i} < \mathbf{n}$  y entonces se tiene que  $\mathbf{i}+1 \leq \mathbf{n}$ .
3. **Base de la inducción:**  $Q(\bar{x}) \wedge B_t(\bar{x}) \Rightarrow R(\bar{x}, \text{triv}(\bar{x}))$ . Se trata de comprobar que para el caso trivial y con parámetros que cumplan precondition la solución que se devuelve cumple postcondición. En el caso trivial es  $\mathbf{i} = \mathbf{n}$  y se devuelve  $\mathbf{y} = \mathbf{0}$ , es evidente que  $0 = \sum_{a=n+1}^n p[\mathbf{a}] \cdot x^{a-1}$ , ya que el dominio del sumatorio se anula, rigurosamente se tiene:  $0 \leq i \leq n \wedge i = n \equiv i = n \Rightarrow 0 = \sum_{a=n+1}^n p[\mathbf{a}] \cdot x^{a-1}$ . En palabras: en el caso trivial el valor devuelto ha de cumplir postcondición
4. **Paso de inducción:**  $Q(\bar{x}) \wedge B_m(\bar{x}) \wedge R(s(\bar{x}), \bar{y}') \Rightarrow R(\bar{x}, c(\bar{y}', \bar{x}))$ . Este suele ser el punto de verificación más complicado en el caso de inmersión no final y suele no entenderse bien. Básicamente y expresado en palabras quiere decir: para parámetros que cumplan precondition, y para caso no trivial, suponiendo que el valor devuelto por la llamada recursiva con parámetros modificados cumple postcondición entonces el valor devuelto por la función tras la operación adicional ha de cumplir también postcondición. En este ejemplo la llamada se realiza incrementando  $\mathbf{i}$ , es decir la función  $s$  transforma  $\mathbf{i}$  a  $\mathbf{i}+1$ , y suponiendo que dicha llamada cumple postcondición el valor devuelto es  $y' = \sum_{a=i+2}^n p[\mathbf{a}] \cdot x^{a-(i+2)}$ . En la operación adicional dicho valor se multiplica por  $x$  y al resultado se le suma el elemento  $\mathbf{p}[\mathbf{i}+1]$ , se ha de verificar que el resultado de dicha operación cumple postcondición. En forma rigurosa:  $0 \leq i \leq n \wedge i < n \wedge y' = \sum_{a=i+2}^n p[\mathbf{a}] \cdot x^{a-(i+2)} \Rightarrow x \cdot y' + p[\mathbf{i}+1] =$
- $$= x \cdot \sum_{a=i+2}^n p[\mathbf{a}] \cdot x^{a-(i+2)} + p[\mathbf{i}+1] = \sum_{a=i+2}^n p[\mathbf{a}] \cdot x^{a-(i+1)} + p[\mathbf{i}+1] = \sum_{a=i+1}^n p[\mathbf{a}] \cdot x^{a-(i+1)}$$
- obsérvese que la condición  $\mathbf{i} < \mathbf{n}$  de caso no trivial garantiza que  $\mathbf{i}+1$  sea un índice válido del vector.
5. **Elección de la estructura de preorden bien fundado.** Se trata de encontrar una función de los parámetros de entrada en los enteros no negativos. Si se toma la función:  $t(p, x, i) = n - i$ , es evidente que siempre toma valor no negativo puesto que

por precondition  $i \leq n$ , la función  $t$  define entonces un preorden bien fundado en el conjunto producto: vectores de tipo vect por reales por naturales, éste es el conjunto dominio de los parámetros de entrada

6. **Demostración de decrecimiento de los datos.** Se ha de demostrar que en cada llamada recursiva la operación previa que se realiza sobre los parámetros de entrada hace decrecer estrictamente los datos en relación al preorden bien fundado previamente definido. De otra forma: que la función  $t(p,x,i)$  toma valores estrictamente decrecientes en cada llamada. Es importante que el decrecimiento sea estricto, en este ejemplo en la llamada recursiva se toma  $i$  incrementado en una unidad y es evidente que:

$$t(p,x,i+1) = n - (i+1) < n - i = t(p,x,i)$$

#### 4.5. Coste

En este ejemplo la reducción de coste se realiza por resta de 1 al valor previo pues  $t(p,x,i+1) = t(p,x,i) - 1$  como se ha comprobado en el apartado anterior. La operación de caso trivial es de coste constante (consiste simplemente en devolver el valor cero) y la operación adicional es también de coste constante (producto por  $x$  y suma del término  $p[i+1]$ ). La recurrencia 1.2 queda:

$$T(n) = \begin{cases} c & \text{si } n < 1 \\ T(n-1) + c & \text{si } n \geq 1 \end{cases}$$

Comparando esta recurrencia con la de 1.2 del texto se tiene para los parámetros  $a, b$  y  $k$  de dicha ecuación los valores 1, 1 y 0 respectivamente, por lo que la solución es (aplicando 1.3 de Peña):

$$T(n) \in \mathcal{O}(n)$$

Es decir, se trata de un algoritmo de coste lineal.

## 5. TRANSFORMACIÓN POR PLEGADO-DESPLEGADO

Se aplicará la técnica de plegado-desplegado para obtener una solución recursiva final a partir de la no final desarrollada en los apartados anteriores, ésta era.

$$\{ Q_1 \equiv 0 \leq i \leq n \}$$

**fun** poli5(p: vect ; x: real; i:natural) **dev** y: real

caso  $i=n \rightarrow 0$

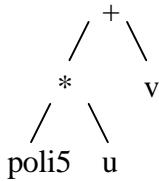
$\dot{y}$   $i < n \rightarrow \text{poli5}(p,x,i+1).x + p[i+1]$

fcaso

ffun

$$\{ R_1 \equiv y = \sum_{a=i+1}^n p[a].x^{a-(i+1)} \}$$

En primer lugar se ha de dibujar el árbol sintáctico de la operación adicional, el resultado de la llamada recursiva se multiplica por la variable x y al resultado de la multiplicación se le suma un término del vector de coeficientes. El árbol sintáctico es:



Se han utilizado las variables u y v para simbolizar los términos que operan con poli5 en la operación adicional.

De aquí se deduce que la nueva función, que se llamará poli6 será de la forma: **poli6(p,x,i,u,v)= poli5(p,x,i)\*u+v**

Las operaciones \* y + tienen elemento neutro, 1 y 0 respectivamente, y son asociativas.

La llamada inicial a poli6 será: poli6(p,x,i,1,0). El plegado-desplegado consiste en:

- 1) Sustituir en **poli6(p,x,i,u,v)=poli5(p,x,i)\*u+v** la función poli5 por su desarrollo, es decir:

$$\text{poli6}(p, x, i, u, v) = \left\{ \begin{array}{l} \text{caso } i = n \rightarrow 0 \\ \text{caso } i < n \rightarrow \text{poli5}(p, x, i + 1).x + p[i + 1] \end{array} \right\} * u + v$$

- 2) Trasladar la operación adicional a cada uno de los casos (desplegado):

$$\text{poli6}(p, x, i, u, v) = \left\{ \begin{array}{l} \text{caso } i = n \rightarrow 0 * u + v \\ \text{caso } i < n \rightarrow (\text{poli5}(p, x, i + 1).x + p[i + 1]).u + v \end{array} \right\}$$

- 3) Aplicando la propiedad de neutro para multiplicación y suma en el caso trivial y la asociativa en el no trivial transformar a:

$$\text{poli6}(p, x, i, u, v) = \left\{ \begin{array}{l} \text{caso } i = n \rightarrow v \\ \text{caso } i < n \rightarrow \text{poli5}(p, x, i + 1).(x.u) + (p[i + 1].u + v) \end{array} \right\}$$

- 4) Teniendo en cuenta que la definición de poli6 es **poli6(p,x,i,u,v)=poli5(p,x,i).u+v**, se puede sustituir la expresión de caso no trivial por su equivalente en términos de poli6 (operación de plegado), aquí el parámetro genérico u toma el valor x.u y el v el valor p[i+1].u+v es (v[i]+r):

$$\text{poli6}(p, x, i, u, v) = \left\{ \begin{array}{l} \text{caso } i = n \rightarrow v \\ \text{caso } i < n \rightarrow \text{poli6}(p, x, i + 1, x.u, p[i + 1].u + v) \end{array} \right\}$$

Y se ha obtenido una solución recursiva final para el problema, ésta solución es:

```

fun poli6(p: vect ; x: real; i:natural; u: real; v: real) dev y: real
    caso i=n → v
    ÿ i<n → poli6(p,x,i+1,x.u,p[i+1].u+v)
    fcaso
ffun

```

La postcondición de esta función será la de poli5 para i=0 es decir:

$$R \equiv y = \sum_{a=1}^n p[\mathbf{a}].x^{a-1}$$

En la precondition se han de incluir condiciones adecuadas para u y v, éstas condiciones pueden deducirse el hecho de que  $\text{poli6}(p,x,i,u,v)=\text{poli5}(p,x,i).u+v$ . Substituyendo las funciones poli6 y poli5 por los predicados postcondición respectivos se tiene:

$$\sum_{a=1}^n p[\mathbf{a}].x^{a-1} = \left( \sum_{a=i+1}^n p[\mathbf{a}].x^{a-(i+1)} \right).u + v$$

que puede escribirse como:

$$\sum_{a=1}^n p[\mathbf{a}].x^{a-1} = \left( \sum_{a=i+1}^n p[\mathbf{a}].x^{a-1} \right)(x^{-i}.u).+v$$

Una solución para u y v en esta ecuación es:  $u = x^i$  y  $v = \sum_{a=1}^i p[\mathbf{a}].x^{a-1}$

La precondition resultante será:

$$Q \equiv 0 \leq i \leq N \wedge u = x^i \wedge v = \sum_{a=1}^i p[\mathbf{a}].x^{a-1}$$

Es interesante revisar con un ejemplo cómo funciona el algoritmo, para n=4 se tiene:

Una llamada inicial: poli6(p,x,0,1,0) que da lugar  
a la llamada poli6(p,x,1,1.x,p[1]+0) que da lugar  
a la llamada poli6(p,x,2,x.x,p[2].x+p[1]) que da lugar  
a la llamada poli6(p,x,3,x<sup>2</sup>.x, p[3]. x<sup>2</sup>+p[2].x+p[1]) que da lugar  
a la llamada poli6(p,x,4, x<sup>3</sup>.x, p[4]. x<sup>3</sup>+p[3]. x<sup>2</sup>+p[2].x+p[1]) , en ésta llamada se alcanza el caso trivial i=4 y se devuelve la variable v, que es:  
p[4]. x<sup>3</sup>+p[3]. x<sup>2</sup>+p[2].x+p[1] . Este resultado se va devolviendo como resultado de las llamadas anteriores y, por tanto, es el valor que devuelve la llamada original.

Obsérvese que en el parámetro u se va calculando x<sup>i</sup> y en el parámetro v se va calculando el subpolinomio de orden i en cada llamada.

Es interesante hacer notar que a esta misma solución se llega si se transforma por plegado-desplegado la segunda solución recursiva no final desarrollada y se aplica luego una inmersión de eficiencia. Puede ser un ejercicio interesante intentarlo.

## 6. TRANSFORMACIÓN RECURSIVO-ITERATIVA

En este apartado se realizará la transformación de la solución recursiva final obtenida en el apartado anterior a iterativa. Para ello se aplicarán la equivalencias expuestas en Peña fig 3.14 respectivamente.

La solución recursiva final era:

```
fun poli6(p: vect ; x: real; i:natural; u: real; v: real) dev y: real
    caso i=n → v
    ÿ i<n → poli6(p,x,i+1,x.u,p[i+1].u+v)
fcaso
ffun
```

y se ejecutaba con llamada inicial poli6(p,x,0,1,0)

Su equivalente iterativa dará lugar a un bucle, en el que:

- La inicialización de variables consistirá en asignar a las variables involucradas los mismos valores que se utilizan en la llamada inicial en la función recursiva. (i:=0,u=1,v=0 ).
- La condición de bucle será la de caso no trivial en el algoritmo recursivo (i<n en el ejemplo)
- Se devolverá tras el bucle el mismo valor que se devolvía en el caso trivial en el algoritmo recursivo (v en el ejemplo)

El invariante del bucle será el predicado dado por la igualdad entre la conjunción de la precondición del algoritmo recursivo y de la función aplicada a los valores iniciales y la propia función para valores genéricos, es decir:

$$P(\bar{x}, \bar{x}ini) \equiv Q(\bar{x}) \wedge f(\bar{x}ini) = f(\bar{x})$$

En este caso:

$$Q \equiv 0 \leq i \leq N \wedge u = x^i \wedge v = \sum_{a=1}^i p[\mathbf{a}].x^{a-1}$$

$$a) P(r, i, 0, 0) \equiv 0 \leq i \leq N \wedge u = x^i \wedge v = \sum_{a=1}^i p[\mathbf{a}].x^{a-1} \wedge poli6(p, x, 0, 1, 0) = poli6(p, x, i, u, v)$$

pero los predicados poli6(p,x,0,1,0) y poli6(p,x,i,u,v) son iguales ya que se trata de la postcondición de suma2 por tanto se puede eliminar de ambos miembros de la igualdad obteniéndose: , en forma más simple y puesto que el invariante es un

predicado que se presupone ha de tomar el valor cierto al inicio de cada vuelta de bucle, se tiene que el invariante es:  $P \equiv 0 \leq i \leq N \wedge u = x^i \wedge v = \sum_{a=1}^i p[\mathbf{a}].x^{a-1}$ .

Obsérvese la técnica para obtener el invariante del bucle equivalente a un algoritmo recursivo final: escribir el predicado dado por la igualdad entre: a) la conjunción formada por la precondition y la postcondición para valores iniciales y b) la postcondición para valores genéricos (ver Peña ecuación 3.19). En la práctica conduce al predicado dado por la precondition de la función recursiva ya que la postcondición del algoritmo recursivo no depende de los parámetros iniciales.

- b) La precondition de la función iterativa será el predicado utilizado en la precondition de la recursiva en el que se sustituyan las variables genéricas por sus valores iniciales es

$$Q_2(xini) = 0 \leq i \leq N \wedge u = x^i \wedge v = \sum_{a=1}^i p[\mathbf{a}]x^{a-1} \Big|_{0,1,0}^{i,u,v} \equiv 0 \leq 0 \leq N \wedge 1 = x^0 \wedge 0 = \sum_{a=1}^0 p[\mathbf{a}].x^{a-1} \equiv \text{ciert}$$

, luego el predicado *cierto* es la precondition del algoritmo iterativo.

A partir de los puntos anteriores se puede escribir el bucle iterativo como:

{ cierto }

**fun** poliiter(p: vect ; x :real ) dev y :real

**var** i:natural; u,v:real;

  i:=0;

  u :=1 ;

  v:=0;

**mientras** i<N **hacer** {  $P \equiv 0 \leq i \leq N \wedge u = x^i \wedge v = \sum_{a=1}^i p[\mathbf{a}].x^{a-1}$  }

    u :=u\*x ;

    v :=v+p[i+1]\*u ;

    i:=i+1;

**fmientras**

**dev** v;

**ffun**

$$\{ R \equiv y = \sum_{a=1}^n p[\mathbf{a}].x^{a-1} \}$$

Obsérvese la esencia de la transformación a iterativo :

- Se inicializan las variables involucradas con los valores que se utilizan en la llamada inicial del equivalente recursivo.
- Se actualizan las variable en el bucle de la misma forma que en la llamada recursiva del equivalente recursivo
- Se acaba el bucle con una condición igual a la trivial del algoritmo recursivo
- Se devuelve un valor igual al de caso trivial del algoritmo recursivo

Puede ser un ejercicio de programación interesante la codificación en MODULA-2 de los diferentes algoritmo. Para ello puede partirse de los ejemplos incluidos en el Ejercicio 1 de esta misma colección.