5. Técnicas básicas de programación de procesos iterativos

En los problemas de programación que estudiaremos de ahora en adelante podemos encontrar un proceso iterativo para resolverlos. Por lo tanto, el diseño de invariantes adecuados es crucial para encontrar un programa que resuelva el problema.

En las secciones que siguen presentaremos varias formas de encontrar invariantes. Una forma de encontrar el invariante es determinando, como ya hemos visto, un proceso iterativo que resuelva el problema y capturar mediante un predicado la esencia de dicho proceso; esta forma es la más intuitiva y requiere de mucha creatividad. Sin embargo, una gran cantidad de problemas de programación pueden ser resueltos aplicando las técnicas que discutiremos en este capítulo, las cuales permiten **derivar** el programa a partir de la manipulación de las pre y post condiciones de una especificación formal de un programa.

Cualquiera sea la forma en que desarrollemos una instrucción iterativa para resolver un problema dado, sea utilizando la intuición o aplicando las técnicas que veremos en este capítulo, es conveniente seguir los siguientes pasos en el desarrollo y en este orden:

1) Determinar el Invariante y las guardias.
2) Inicialización: Establecer el invariante la primera vez. Es decir, establecer el valor inicial de las variables para que se cumpla el invariante.
3) Determinar la función de cota a partir de (1) y (2).
4) Desarrollar el trozo de programa que corresponde al cuerpo de cada guardia de manera que cumpla la especificación: {Invariante y Guardia} S { Invariante }. Este paso se puede descomponer en primero modificar la función de cota como última instrucción de S y en función de esto desarrollar el resto de S. La modificación de la función de cota es la que garantiza que la función decrece (o crece) estrictamente.

5.1. Técnica: Eliminar un predicado de una conjunción

Tenemos el problema siguiente: Calcular, utilizando sólo sumas, restas y multiplicaciones, el cociente y el resto de la división entera de A entre B, donde A y B son números enteros, $A \geq 0$ y $B > 0$.

La especificación formal es:

```
[  const  A, B: entero;
   var  q, r: entero;
   {  A ≥ 0   ∧   B > 0  }
    divmod
   {  q = A div B  ∧  r = A mod B  }
]
```

Usando la definición de div y mod, obtenemos una especificación mas refinada:

```
[  const  A, B: entero;
```

```
  var  q, r: entero;
  {  A ≥ 0   ∧   B > 0  }
   divmod
  {  A = q * B + r   ∧  0 ≤ r   ∧   r <  B }
]
```

La postcondición puede ser manipulada de la siguiente forma:

```
        A = q * B + r   ∧  0 ≤ r   ∧   r <  B
≡  aritmética (despejando r)
        r =  A  - q * B    ∧  0 ≤ r   ∧   r <  B
≡  lógica (sustitución de iguales)
        r =  A  - q * B    ∧  0 ≤ A  - q * B   ∧  A  - q * B <  B

≡  aritmética (sumando B en la segunda fórmula)
        r =  A  - q * B    ∧  B ≤ A  - q * B + B   ∧  A  - q * B <  B
≡  aritmética (sacando factor común)
        r =  A  - q * B    ∧  B ≤ A  - (q – 1) * B   ∧  A  - q * B <  B
```

Entonces, encontrar un q y un r que satisfacen la postcondición anterior es equivalente a hallar un q (entero no negativo) tal que $0 \le A - q*B < B$ y esto a su vez es equivalente a hallar un entero no negativo q tal que $A - q*B < B$ y $A - (q-1)*B \ge B$. Y r será igual a $A - q*B$. Es decir, buscar un q tal que al restarle a A la cantidad $(q-1)*B$, el resultado (llamémoslo **resta**) es mayor o igual que B, pero al restarle de nuevo B a **resta** ($A - q*B = A - (q-1)*B - B$), el resultado da menor que B.

Por lo tanto nuestro problema lo podemos especificar también como:

```
[  const  A, B: entero;
   var  q, r: entero;
   {  A ≥ 0   ∧   B > 0  }
    divmod
   {  A – q*B <  B  ∧  A – (q-1)*B  ≥  B  ∧  r = A – q*B }
]
```

Si existe un q que satisface la postcondición entonces éste sólo puede tomar valores entre 0 y A, ambos inclusive. Por lo que bastará con recorrer los números naturales entre 0 y A para encontrar dicho número. Veamos por qué.

El primer predicado de la postcondición, $A - q*B < B$, implica que 0 es cota inferior de q:

```
        A – q*B <  B
≡   aritmética (sumando q*B a ambos lados y sacando factor común)
        A < (q+1)*B
⇒   como A ≥ 0 por precondición
```

$0 < (q+1)*B$

≡ aritmética (pues $B > 0$ por precondición)

$0 < (q+1)$

≡ restando 1 a ambos lados

$-1 < q$

≡ aritmética (q es entero y mayor estricto que -1)

$0 \leq q$

El segundo predicado en la postcondición, $A - (q-1)*B \geq B$, implica que q tiene como cota superior a A:

$A - (q-1)*B \geq B$

≡ aritmética (restando B a ambos lados y sacando factor común)

$A - q*B \geq 0$

≡ aritmética

$A \geq q*B$

⇒ como $B > 0$ (precondición), se tiene que $B \geq 1$ y así $q*B \geq q$

$A \geq q$

Por lo tanto hemos demostrado que:

$$A - q*B < B \wedge A - (q-1)*B \geq B \Rightarrow 0 \leq q \wedge q \leq A$$

Un proceso iterativo que resuelve el problema es recorrer con la variable q los números naturales desde 0 hasta A (vimos que q no puede ser mayor que A y alcanza el valor de A con B=1 y r=0). Y partiendo de q=0, mientras $A - q*B$ sea mayor o igual que B, sumar 1 a q. El número de iteraciones del proceso iterativo está acotado por A.

Note que al concluir el proceso iterativo tendremos q = A div B. Y podemos entonces asignar a r el valor $A - q*B$. Note también que al comienzo de cada iteración (incluyendo la última, cuando la condición de continuación no se satisface) se cumple que $A - (q-1)*B \geq B$, y este captura la esencia del proceso.

Así nuestro programa sería:

```
[  const  A, B: entero;
   var  q, r: entero;
   {  A ≥ 0   ∧   B > 0  }
   q := 0;
   do A – q*B ≥  B  →  q :=  q + 1
   od;
    r := A – q*B
   {  A – q*B <  B  ∧  A – (q-1)*B  ≥  B  ∧  r = A – q*B }
]
```

Ejercicio: muestre que el programa anterior es correcto utilizando el invariante $A - (q-1)*B \geq B$ y la función de cota creciente q (o función de cota decreciente A-q).

Para el problema anterior podemos utilizar una técnica para derivar el programa, conocida como **la técnica de eliminación de un predicado cuando la postcondición se expresa como una conjunción de predicados.** Esta técnica expresa lo siguiente:

Cuando la postcondición R es de la forma $P \wedge Q$, uno puede tratar de tomar uno de los predicados como el invariante y el otro como la negación de la guardia de la instrucción iterativa, es decir:

$\{ P \}$ do $\neg Q \rightarrow S$ od $\{ P \wedge Q \}$

```
[  const  A, B: entero;
   var  q, r: entero;
   { A ≥ 0  ∧  B > 0 }
    divmod
   { R: A – q*B <  B  ∧  A – (q-1)*B  ≥  B  ∧  r = A – q*B }
]
```

Tomemos como invariante a P: $A - (q-1)*B \geq B \wedge r = A - q*B$, y a $A - q*B \geq B$ como guardia. Lo cual resulta en un programa de la forma:

$\{ P \}$ do $A - q*B \geq B \rightarrow S$ od $\{ R \}$

El invariante puede ser establecido inicialmente mediante las asignaciones q := 0 y r := A. Como P implica $A - q*B \geq 0$ y debe decrecer en cada iteración hasta llegar a $A - q*B < B$, podemos tomar a $A - q*B$ como función de cota. En cada iteración podemos aumentar en 1 a q para que decrezca $A - q*B$ y para mantener el invariante habría que reducir en B a r. Esto lleva al siguiente programa **divmod:**

```
[  const  A, B: entero;
   var  q, r: entero;
   { A ≥ 0  ∧  B > 0 }
   q := 0;  r := A;
   do A – q*B  ≥  B →   r, q :=  r – B, q := q +1
   od
   { R: A – q*B <  B  ∧  A – (q-1)*B  ≥  B  ∧  r = A – q*B }
]
```

Note además, que al ser $r = A - q*B$ un invariante, podemos reemplazar la expresión $A - q*B$ en el programa anterior por r, lo cual mejora la eficiencia del programa al no tener que calcular cada vez la guardia. Y el invariante queda: $r \geq 0 \wedge r = A - q*B$. quedando el programa con anotaciones siguiente:

```
[  const  A, B: entero;
```

```
  var  q, r: entero;
  {  A ≥ 0  ∧  B > 0  }
  q := 0;  r := A;
  { Invariante P: r  ≥  0  ∧  r = A – q*B, demo 0, función de cota decreciente: r }
  do  r  ≥  B  →  { P ∧  r  ≥  B }  r, q :=  r – B; q := q +1  { P, demo 1 }
  od
  {  R: r <  B  ∧  r  ≥  0  ∧  r = A – q*B, demo 2, terminación: demo 3 }
]
```

Hagamos las demostraciones demo 0, demo1, demo 2 y demo 3 para demostrar la correctitud del programa anterior.

<u>Demo 0</u>:

Mostremos que se cumple:

```
  {  A ≥ 0  ∧  B > 0  }
  q := 0;  r := A
  { r  ≥  0  ∧  r = A – q*B}
```

En efecto:

```
        ( r  ≥  0  ∧  r = A – q*B ) (r := A) (q := 0)
≡    sustitución
        A  ≥  0  ∧  A = A – 0*B
≡  aritmética
        A  ≥  0  ∧  A = A
≡  aritmética
        A ≥ 0
⇐       lógica ( p ∧ q ⇒ p )
        A ≥ 0  ∧  B > 0
```

<u>Demo 1</u>:

Demostremos que se cumple { P ∧ r ≥ B } r, q := r – B, q +1 { P }

```
        P( r, q :=  r – B, q+1)
≡ sustitución
        r - B  ≥  0  ∧  r – B  = A – (q+1)*B
≡ aritmética
        r  ≥  B  ∧  r  = A – q*B
⇐ por lógica ( p ∧ q ⇒ p )
        r  ≥  B  ∧  r  = A – q*B  ∧  r  ≥  0
```

Demo 2:

Es directa.

Demo 3 (terminación):

Dada la función de cota r, sabemos por el invariante que $r \geq 0$ al comienzo de cada iteración.

Debemos mostrar ahora que si se tiene r = C al comienzo de una iteración y se cumple la guardia entonces al concluir la iteración se tendrá r < C. En efecto, supongamos que r = C. La instrucción correspondiente a la guardia resta B a r, por lo que al concluir la instrucción r será igual a C-B, y como B es positivo, el nuevo valor de r será menor estricto que el original.

Hemos podido aplicar directamente la **técnica de eliminación de un predicado de la conjunción**, a la especificación:

[const A, B: entero;
  var q, r: entero;
  { $A \geq 0 \ \wedge \ B > 0$ }
  divmod
  { R: $A = q*B + r \wedge r \geq 0 \wedge r < B$ }
]

Tomando como invariante a P: $A = q*B + r \wedge r \geq 0$, y $r \geq B$ como la guardia. Lo cual resulta en un programa de la forma:

{ P } do $r \geq B \rightarrow$ S od { R }

El invariante puede ser establecido inicialmente mediante las asignaciones q := 0 y r := A. Como P implica $r \geq 0$ y r debe decrecer en cada iteración hasta llegar a $r < B$, podemos tomar a r como función de cota. En cada iteración podemos reducir r en B y para mantener el invariante habría que aumentar q en 1. Esto lleva al siguiente programa **divmod:**

[const A, B: entero;
  var q, r: entero;
  { $A \geq 0 \ \wedge \ B > 0$ }
  q := 0; r := A;
  do $r \geq B \rightarrow$ r, q := r – B, q+1
  od
  { $A = q*B + r \ \wedge \ r \geq 0 \wedge r < B$ }
]

El programa anterior es exactamente el mismo que obtuvimos antes.

Hemos podido tomar como invariante cualquier combinación de conjunciones y tratar de derivar el programa. Es posible que no tengamos éxito con algunas combinaciones.

Ejercicio: Utilice la técnica anterior tomando como invariante a: $A - q*B < B$, para derivar el programa dado por la especificación siguiente (este programa determina el cociente q de la división entera de A entre B y sólo debe usar sumas y restas):

[const A, B: entero;
  var q: entero;
  { $A \geq 0 \ \wedge \ B > 0$ }
  división entera
  { $A - (q-1)*B \geq B \ \wedge \ A - q*B < B$ }
]

Ejemplo: Cálculo de la raíz cuadrada entera de un número entero no negativo.

Problema: Dado un número entero no negativo N, se quiere determinar la raíz cuadrada entera de N, es decir, el mayor número entero no negativo que multiplicado por el mismo es menor o igual a N.

Este problema es "del mismo tipo" que el anterior, en el sentido que hay que recorrer el conjunto de los números naturales, partiendo desde cero hasta un cierto número, hasta encontrar un número x que cumpla una determinada propiedad P(x) y el sucesor del número, x+1, no cumpla la propiedad (se cumple $\neg P(x+1)$). En el ejemplo anterior (primera versión de desarrollo) la propiedad era P(q): $A - (q-1)*B \geq B$.

En el ejemplo que nos ocupa, queremos hallar un número entero no negativo x tal que $x^2 \leq N$ y $(x+1)^2 > N$, donde la propiedad P(x) es $x^2 \leq N$.

La especificación formal es:

[const N: entero;
  var x: entero;
  { $N \geq 0$ }
  raíz cuadrada
  { $x^2 \leq N \ \wedge \ (x+1)^2 > N$ }
]

Como N es no negativo, podemos recorrer los números naturales partiendo desde cero hasta encontrar el número x que cumpla $x^2 \leq N$ y $(x+1)^2 > N$. Note que x nunca podrá exceder a N (si $x > N$ entonces por ser $N \geq 0$ se tiene que $x \geq 1$ y así $x^2 \geq x > N$, es decir, x no cumple con $x^2 \leq N$). Por lo tanto en el proceso iterativo que consiste en recorrer los números naturales partiendo desde cero hasta encontrar el número buscado siempre se ha de cumplir el predicado $x^2 \leq N$ (este sería el invariante) y la condición de continuación del proceso iterativo deberá ser $(x+1)^2 \leq N$. Note que $x = 0$ satisface inicialmente el invariante.

Como $x^2 \leq N$ es equivalente a $N - x^2 \geq 0$, podemos tomar como función de cota decreciente a $N - x^2$, sin embargo, $N - x^2$ decrece con un incremento de x si y sólo si $x \geq 0$, y esto no puede ser inferido del invariante y la guardia: $x^2 \leq N \wedge (x+1)^2 \leq N$. Por lo tanto debemos reforzar el invariante a: $x \geq 0 \wedge x^2 \leq N$.

Por lo tanto el programa es:

```
[ const  N: entero;
  var x: entero;
  { N ≥ 0 }
  x := 0;
  { Invariante P: x ≥ 0 ∧ x² ≤ N, demo 0, función de cota decreciente: N – x² }
  do  (x+1)*(x+1) ≤ N  →  x  :=  x+1
  od
  {  x² ≤ N  ∧  (x+1)² > N  }
]
```

Note que otra solución iterativa al problema de hallar un entero x que cumpla P(x) y $\neg P(x+1)$, es partir de un número entero x que cumpla $\neg P(x+1)$ y luego ir disminuyendo x en 1 hasta encontrar un x que cumpla $P(x) \wedge \neg P(x+1)$.

Ejercicios:

1) demuestre la correctitud del programa anterior.
2) Ejercicios página 56 del Kaldewaij.
3) Aplique **técnica de eliminación de un predicado de la conjunción** a la especificación:

```
[ const  N: entero;
  var x: entero;
  { N ≥ 0 }
    raíz cuadrada
  {  x² ≤ N  ∧  (x+1)² > N  }
]
```

utilizando como invariante a $x^2 \leq N$, luego utilizando como invariante a $(x+1)^2 > N$

5.2. Técnica: reemplazo de constantes por variables

Problema: queremos calcular A^B, con B entero no negativo, donde, por definición, $A^0 = 1$, para todo A (incluyendo 0).

Este problema se especifica formalmente como sigue:

```
[  const  A, B:entero;
   var r: entero;
   {  B ≥ 0 }
```

exponenciación
{ r = A^B }
]

Un proceso iterativo que resuelve este problema es: en cada iteración multiplicar a r (partiendo de r = 1) por A y guardar el resultado en r. Más formalmente, al comienzo de la iteración i (comenzando desde la iteración 0) la variable r contendrá A^i, por lo que un invariante sería r = A^i. Note que si al comienzo de la iteración 0 (i=0) se deberá tener r =1. El esquema del programa con la fase de inicialización sería:

r , i := 1, 0;
{ Invariante: r = A^i}
do B $\rightarrow$ S
od
Como el número de la iteración i cumple con $0 \leq i \leq B$, podemos incorporar este predicado al invariante. Por otro lado, la condición de continuación del proceso iterativo debe ser i < B.

r , i := 1, 0;
{ Invariante: r = $A^i \wedge 0 \leq i \leq B$}
do i < B $\rightarrow$ S
od

Note que para que al comienzo de la iteración i+1 la variable r contenga A^{i+1}, basta con asignar r*A a r en la iteración i, e incrementar en 1 la variable i para que refleje la siguiente iteración:

r , i := 1, 0;
{ Invariante: r = $A^i \wedge 0 \leq i \leq B$}
do i < B $\rightarrow$ r, i := r*A, i+1
od

Por lo tanto, al comienzo de la iteración B la variable r contendrá el resultado y debe concluir el proceso iterativo. Una función de cota estrictamente creciente entre dos iteraciones sucesivas es i (el número de la iteración), la cual está acotada superiormente por B. Podemos tomar también como función de cota a B-i (estrictamente decreciente entre dos iteraciones sucesivas) y acotada inferiormente por 0.

El programa es:

[const A, B:entero;
var r, i: entero;
{ P: $B \geq 0$ }
r, i := 1, 0;
{Invariante I: r = $A^i \wedge 0 \leq i \leq B$, función de cota creciente: i }
do i < B $\rightarrow$ r, i := r*A, i+1

{ Q: $r = A^B$ }
]

Para demostrar la correctitud del programa anterior, hay que demostrar:

1) Se cumple: { P } r, i := 1, 0; { I }
2) Se cumple: { $I \wedge i < B$ } r := r*A; i := i+1{ I }
3) [$I \wedge i \geq B \Rightarrow Q$]
4) Se cumple: { $I \wedge i = C$ } r := r*A; i := i+1 { $i < C$ }

Mostremos (2):

I(r, i := r*A, i+1)
≡ sustitución
$r*A = A^{i+1} \wedge 0 \leq i+1 \leq B$

Suponiendo que se cumple $I \wedge i < B$, es decir, $r = A^i \wedge 0 \leq i \leq B \wedge i < B$, tenemos que multiplicando por A en ambos lados de $r = A^i$ tenemos $r*A = A^{i+1}$. Por otro lado al ser $i \geq 0$ entonces se tiene que $i+1 \geq 0$ y al ser $i < B$ se tiene que $i+1 \leq B$.

La **técnica de reemplazo de constantes por variables** nos permite derivar el mismo programa anterior estableciendo un invariante a partir de la postcondición mediante el reemplazo de una constante de la postcondición por una variable nueva. Las posibilidades son las siguientes:

$$r = x^B, \quad r = A^x, \quad r = x^y$$

Utilicemos como invariante a

P_0: $r = A^x$

Entonces $P_0 \wedge x = B$ implica la postcondición, y P_0 puede ser establecida con la inicialización r, x :=1, 0.

Siempre es conveniente acotar a las variables nuevas que introduzcamos. Podemos fijar una cota superior para x, y agregar el predicado siguiente al invariante:

P_1: $x \leq B$

Esto lleva al esquema de programa:

$$R, x := 1, 0 \ \{ P_0 \wedge P_1 \} ; \textbf{do } x \neq B \rightarrow S \textbf{ od } \{ r = A^B \}$$

Como x inicialmente es cero y deseamos llegar a x = B, debemos investigar el efecto de incrementar en 1 a x como última instrucción de S:

$(P_0 \wedge P_1)(x := x+1)$

$\equiv$ por sustitución

$r = A^{x+1} \wedge x + 1 \leq B$

Si a S lo representamos por la secuenciación S1; x:= x+1, deberíamos tener:

(1) $\{ P_0 \wedge P_1 \wedge x \neq B\}$ S1 $\{ r = A^{x+1} \wedge x + 1 \leq B \}$; x := x+1 $\{ P_0 \wedge P_1 \}$

Si P_0 se cumple tenemos que $A^{x+1} = A^* A^x = A * r$, por lo que S1 sería la instrucción r := A * r, además, $P_1 \wedge x \neq B$ no son afectados por la asignación a r e implican $x < B$, y esto último implica $x+1 \leq B$.

Así, el programa exponenciación es:

```
[  const  A, B:entero;
   var r, x: entero;
   {  A ≥ 0  ∧  B ≥ 0 }
   r, i := 1, 0;
   {Invariante: r = A^x  ∧  x ≤B, función de cota decreciente: B-x }
   do  x ≠ B  →  r, x  := A*r, x+1
   {  r = A^B  }
]
```

Ejercicios:

1) Hacer un programa para los ejercicios 12, 13, 15, 16, 17, 24, 25, 26, 30 de la sección 1 del problemario de Michel Cunto.
2) Dibujar N círculos concéntricos con la máquina de trazados, con centro (0,0) y los radios siguen la progresión aritmética 1+2*i, para los círculos 0,1,...N-1

Problema: sumar los elementos de una secuencia de enteros de largo N.

La especificación es:

```
[  const  N: entero;
   const f: secuencia de enteros;
   var  x: entero;
   {  |f| = N  ∧  N  ≥  0 }
   suma
   {  x = (∑j: 0 ≤ j < N : f [j] ) }
]
```

En este ejemplo utilizaremos el tipo de datos “secuencia de enteros” y expresaremos la solución del problema en términos de las operaciones de observación de los elementos de una secuencia.

Un proceso iterativo que resuelve este problema es muy parecido al que describimos en la sección 4.5, la suma de los cuadrados de los primeros N números naturales. El proceso consiste en ir acumulando en la variable x la suma de los elementos de la secuencia. Al comienzo de la iteración i (comenzando con la iteración 0) se han sumado los elementos de la secuencia entre 0 y i-1. El proceso termina en la iteración N. Por lo tanto, un invariante para este proceso es: $(x = (\sum j: 0 \leq j < i : f [j])) \wedge (0 \leq i \leq N)$. En cada iteración, menos en la última, sumamos el elemento f[i] a la variable x. Sabemos que $(\sum i: 0 \leq i < 0 : f [i])$ es igual a cero (la suma de cero números es cero), y entonces inicialmente la variable x es cero. El número de la iteración i sirve como función de cota creciente.

El programa sería:

```
[  const  N: entero;
   const f: secuencia de enteros;
   var  x,i: entero;
   {  |f| = N  ∧  N  ≥  0 }
   x , i := 0, 0;
   { Invariante I: ( x = ( ∑j: 0 ≤ j < i : f [j] ) )  ∧  (0 ≤ i ≤ N),
                                                  función de cota creciente: i }
   do  i < N  →  x, i := x +f[i], i+1  od
   {  x = (∑j: 0 ≤ j < N : f [j] ) }
]
```

Ejercicio: demuestre la correctitud del programa anterior.

Podemos aplicar la **técnica de reemplazo de constantes por variables** para derivar un invariante y obtener el mismo programa anterior. Reemplazamos una constante de la postcondición $x = (\sum j: 0 \leq j < N : f [j])$ por una variable nueva. El cuantificador que aparece en esta postcondición posee dos constantes: 0 y N. Reemplacemos N por una variable i y propongamos como invariante a:

$$P_0: \quad x = (\sum j: 0 \leq j < i : f [j])$$

Por lo tanto $P_0 \wedge i=N$ implica la postcondición. La suma sobre un rango vacío es cero, por lo que inicialmente se puede establecer P_0 con la instrucción x, i := 0, 0. El programa que buscamos tiene el esquema siguiente:

$$x, i := 0, 0;\ \text{do } i \neq N \rightarrow S \text{ od}$$

Debemos encontrar S. Supongamos que se cumple $P_0 \wedge i \neq N$, es decir, se cumple $x = (\sum j: 0 \leq j < i : f [j]) \wedge i \neq N$. Este predicado refleja el estado de las variables justo antes de ejecutarse S. Como el interés es incrementar i hasta llegar a N, veamos lo que significa un incremento de i en 1 en la expresión que refleja la suma en P_0:

$(\sum j: 0 \leq j < i+1 : f [j])$

= siempre y cuando $i \geq 0$ podemos separar (habría que agregar este predicado a P_0

como invariante)
$(\sum j: 0 \le j < i : f[j]) + f[i]$
= como $x = (\sum j: 0 \le j < i : f[j])$ por P_0
$x + f[i]$

Por lo tanto basta con asignar a x el valor de la expresión x + f[i] para que se cumpla $P_0(i := i+1)$, es decir, $x = (\sum j: 0 \le j < i+1 : f[j])$. Tenemos entonces que se cumple:

$$\{ P_0 \wedge 0 \le i \wedge i \neq N \}\ x := x+f[i]\ \{ P_0(i := i+1) \}\ ;i := i+1\ \{ P_0 \}$$

Una función de cota apropiada es N-i (también sirve i, creciente estricta). Esta sería estrictamente decreciente pues i aumenta en 1 en cada iteración. Sin embargo para la prueba de terminación necesitaremos que siempre se cumpla $i \le N$, lo que garantiza que la función de cota está acotada inferiormente por cero.

Por lo tanto, haber hecho el reemplazo de una constante por una variable en la postcondición nos llevó a encontrar los siguientes invariantes:

P_0: $x = (\sum j: 0 \le j < i : f[j])$
P_1: $0 \le i \le N$

Finalmente el programa es:

```
[  const  N: entero;
   const f: secuencia de enteros;
   var  x,i: entero;
   {  |f| = N  ∧  N  ≥  0 }
   x , i := 0, 0;
   { Invariante I: ( x = ( ∑j: 0 ≤ j < i : f [j] ) )  ∧  (0 ≤ i ≤ N), cota: i }
   do  i ≠ N  →  x, i := x +f[i], i+1  od
   {  x = (∑j: 0 ≤ j < N : f [j] ) }
]
```

Ejercicios:

1) Aplicando la técnica de derivación de reemplazo de constantes por variables, reemplace la constante 0 por n en la postcondición del ejemplo anterior con el objeto de hallar un invariante y derive el programa correspondiente.

5.3. Técnica: fortalecimiento de invariantes

Problema: queremos determinar el N-ésimo número de Fibonacci, fib(N). Los números de Fibonacci se definen de la siguiente forma:

fib(0) = 0, fib(1) = 1, y fib(n+2) = fib(n) + fib(n+1) para $n \ge 0$.

La especificación formal del programa sería:

[const N: entero;
  var x: entero;
  { $N \geq 0$ }
  Fibonacci
  { x = fib(N) }
]

Aplicando la técnica de reemplazo de una constante por una variable en la postcondición, si reemplazamos a N por i, podemos proponer como invariante a "x = fib(i)". Este invariante nos dice que al comienzo de la iteración i se tiene x = fib(i). Vemos que $x = fib(i) \wedge i = N$ implican la postcondición, por lo que la guardia sería $i \neq N$. El invariante se puede establecer inicialmente asignando a x el valor fib(0) (= 0) y asignar 0 a i . Más aún, tenemos como invariante también a $0 \leq i \leq N$. Y una función de cota decreciente: N-i.

El esquema del programa es:

x , i := 0, 0;
do $i \neq N \rightarrow S$ od

Determinemos S. Note que al comienzo de la iteración i+1 la variable x debe contener fib(i+1). Suponiendo que al comienzo de la iteración i se cumple $x = fib(i) \wedge 0 \leq i \leq N \wedge i \neq N$ (es decir, el invariante y la guardia), no tenemos forma de calcular fib(i+1) (que será el valor de x al concluir la iteración i) a partir sólo de x y de i. Por lo que introducimos una nueva variable z que al comienzo de la iteración i contenga fib(i+1). De esta forma **fortalecemos nuestro invariante (aquí estamos aplicando la técnica de fortalecimiento del invariante)** y éste será:

$$x = fib(i) \wedge z = fib(i+1) \wedge 0 \leq i \leq N$$

Que puede ser establecido inicialmente asignando 0, 1, 0 a x, z, i respectivamente. El nuevo esquema de programa sería:

x , z, i := 0, 1, 0;
do $i \neq N \rightarrow S$ od

Para que se mantenga el invariante, como estamos analizando un proceso iterativo, al comienzo de la iteración i+1 la variable x deberá contener fib(i+1) y la variable z deberá contener fib(i+2). En la iteración i se deberá colocar en x el valor fib(i+1) (= valor inicial de z al comienzo de la iteración i) y en la variable z se deberá asignar el valor fib(i+2) (= fib(i) + fib(i+1) = valor inicial de x + valor inicial de z al comienzo de la iteración i).

Por lo que el programa sería:

[const N: entero;

```
  var x, y: entero;
  { N ≥ 0 }
  x , z, i := 0, 1, 0;
  do i ≠ N → "asignar a x el valor inicial de z, y asignar a z la suma de los valores iniciales
                de x, z";
                i := i + 1
  od
  { x = fib(N) }
]
```

Otra manera de razonar para deducir el trozo de programa "asignar a x el valor inicial de z, y asignar a z la suma de los valores iniciales de x, z", sería como sigue:

Sean P_0: x = fib(i) , P_1: z = fib(i+1), R: $0 \le i \le N$. Suponiendo que se cumple $P_0 \wedge P_1 \wedge R \wedge i \neq N$, un incremento de i en 1 nos debería llevar a que se cumpla $(P_0 \wedge P_1 \wedge R)$ (i := i+1) y esto es equivalente a $x = fib(i+1) \wedge z = fib(i+2) \wedge 0 \le i+1 \le N$. Asignando el valor de z a x queda establecido x = fib(i+1). El predicado $0 \le i+1 \le N$ queda directamente establecido pues se tiene $i \neq N \wedge 0 \le i \le N$. El predicado z = fib(i+2) queda establecido por:

```
        fib(i+2)
=  definición de fib
        fib(i) + fib(i+1)
= por P0 y P1
        x + z
```

Por lo tanto, el trozo de programa "asignar a x el valor inicial de z, y asignar a z la suma de los valores iniciales de x, z" lo podemos implementar con una asignación múltiple x, z := z, x + z, o introduciendo una nueva variable t que conserve el valor original de una de las dos variables x, z: t := x; x:= z; z:= t + z

Finalmente el programa con sus anotaciones sería:

```
[ const N: entero;
  var x, z, t: entero;
  { N ≥ 0 }
  x , y, i := 0, 1, 0;
  { Invariante: x = fib(i) ∧ z = fib(i+1) ∧ 0 ≤ i ≤ N, demo 0;
     función de cota decreciente: N-i }
  do i ≠ N → { x = fib(i) ∧ 0 ≤ i ≤ N ∧ i ≠ N}
                x, z := z, x + z;
                i := i+1
                { x = fib(i) ∧ 0 ≤ i ≤ N, demo 1 }
  od
  { x = fib(N), demo 2, terminación: demo 3 }
]
```

Ejercicios:

1) Hacer un programa que calcule (max p: $0 \leq p \leq N$: ($\sum$j: $p \leq j < N$: s[j])), para $N \geq 0$ y s una secuencia de largo N. Ayuda: trate de simplificar la expresión y aplicar las técnicas de reemplazo de constantes por variables y aplicar la regla de fortalecimiento del invariante reemplazando en la postcondición la constante 0 por una variable.

2) Desarrolle un programa correcto que resuelva los siguientes problemas (trate de utilizar, de ser posible, las técnicas dadas de derivación):

   A. Calcular ($\sum$i: $0 \leq i < N$: i^3).
   B. Calcular ($\sum$i: $0 \leq i < N$: s[i]). Donde s es una secuencia de largo N.
   C. Calcular ($\prod$i: $0 \leq i < N$: 1/(i+1)).
   D. Calcular el mayor elemento de una secuencia s de números enteros de largo n.

Páginas 62 y 71 de Kaldewaij, reemplazando la palabra “arreglo” por “secuencia”.