Programación

Asignatura 240205 Programación. Curso 2020–2021

Lectura[5]: **Derivación de programas iterativos**

Dr. José Ramón González de Mendívil
mendivil@unavarra.es
Departamento de Estadística, Informática y Matemáticas
Edificio Las Encinas
Universidad Pública de Navarra

Resumen

La derivación de programas iterativos consiste en obtener el programa solución a partir de la especificación del problema algorítimico utilizando para ello, una propuesta de invariante y el método de verificación, del bucle correspondiente, como una herramienta para calcular las sentencias desconocidas del programa. Los programas obtenidos, en forma de algoritmos, mediante derivación, son correctos, y en ocasiones, mucho más eficientes que aquellos que podemos obtener usando un método de diseño basado en prueba y error. En esta Lectura se presenta el método de derivación utilizando ejemplos de distinto grado de dificultad.

Índice

1. Introducción

En la mayor parte del resto del curso de Programación nos vamos a dedicar a derivar algoritmos (programas en pesudocódigo) que cumplan una determinada especificación (problema algorítmico). Las reglas de inferencia que hemos estudiado en la Lectura[4], para las diferentes formas de componer sentencias, nos sirven para verificar si un determinado programa, ya establecido, cumple con su especificación; pero dichas reglas se pueden emplear también como un cálculo de las sentencias que desconocemos a la hora de abordar el diseño de la solución. Eso es justamente el propósito de la **derivación**: a partir del conocimiento de la especificación, obtener la solución, en forma de algoritmo, que la cumple.

Los problemas que vamos a abordar tienen soluciones que involucran a una o varias sentencias **mientras**...**fmientras** y ya hemos indicado que el predicado invariante es crucial para la verificación de tales programas. Por tanto, disponer de invariantes adecuados también es crucial para la derivación de las soluciones correctas. A lo largo del curso veremos que hay diferentes formas de deducir los predicados invariantes a partir de la postcondición del problema. La derivación de un programa no es una tarea completamente mecánica y en general, require cierta creatividad. No obstante, muchos programas típicos que ocurren en la programación pueden resolverse mediante cálculo lógico, aplicando los métodos que vamos a estudiar en esta Lectura[5].

Un mismo problema algorítmico puede tener diferentes programas que lo resuelven. La cuestión es, ¿cómo se comparan dichas soluciones?. Normalmente utilizamos la notación $\mathcal{O}$ que indica el orden del peor caso del coste computacional en el tiempo: su orden de complejidad en tiempo. Esto se refiere a que, independiente del tiempo 'real' que duran las sentencias, se mide el número de sentencias ejecutadas en el peor caso y se busca la función más sencilla que acota dicho número. Por ejemplo, considera que al medir el peor caso has obtenido $\frac{1}{2}N^2 - 2N + 4$, para un 'tamaño del problema' N. Entonces decimos que el programa tiene una complejidad de orden $\mathcal{O}(N^2)$. Sin entrar en mucho detalle, a partir de un N_0 dado, hay una constante c tal que para todo $N > N_0$, se cumple $c\,N^2 > \frac{1}{2}N^2 - 2N + 4$. Los programas que no tienen bucles tienen una complejidad constante. Los programas iterativos (con bucles) tienen diferentes grados de complejidad en el tiempo. La **eficiencia** (en el tiempo) de un programa es de gran importancia en Informática puesto que los programas se escriben una vez y se ejecutan muchas veces![1]

Según lo indicado, un programa A es más eficiente (en tiempo) que un programa B para el mismo problema si el orden de complejidad de A es menor que el de B. Por ejemplo, el programa A tarda 15 minutos en una máquina M1 en resolver sobre unos datos ($N = 1000$) un problema dado, y es de orden $\mathcal{O}(N)$. El programa B es de orden $\mathcal{O}(N^2)$, luego sobre la misma máquina M1 y los mismos datos tardará más o menos 300 horas. Si un programador resuelve el mismo problema con un algoritmo C de orden $\mathcal{O}(log(N))$, sobre la misma máquina M1 y los mismos datos su programa tardará más o menos 10 segundos. Una mejora realmente significativa se obtiene derivando una solución completamente diferente y no meramente ahorrando instrucciones.

Para nuestra desgracia hay problemas que sus soluciones tienen complejidades muy altas, de orden exponencial $\mathcal{O}(2^N)$. Se les suele denominar 'problemas duros': tienen solución pero es de complejidad exponencial. El estudio de este tipo de problemas y cómo se abordan en la actualidad es parte de la teoría de Complejidad computacional. La mayor parte de los problemas que implican una optimización suelen ser de ese tipo. Por otra parte, es interesante que existan este tipo de problemas para que la criptográfia tenga su lugar en la Informática: básicamente encriptar unos datos mediante claves de manera que la búsqueda de las claves sea un problema intratable.

[1] La eficiencia en memoria de un programa es el orden del peor caso de la cantidad de almacenamiento que se necesita para su ejecución. También es un factor a tener en cuenta para comparar diferentes soluciones, pero es menos crítico. El almacenamiento (memoria) se puede reutilizar pero el tiempo no.

2. Cuestiones sobre la notación

Sobre sustituciones: Cuando hacemos la sustitución en un predicado P ponemos $P(x := exp)$ para construir el nuevo predicado en donde toda aparición de x en P se sustituye por la expresión Exp. Normalmente usaremos la notación $(P)^x_{Exp}$ para indicar la misma sustitución, $[[\,(P)^x_{Exp} \equiv P(x := exp)\,]]$. Esto es un poco más cómodo de escribir cuando tenemos que hacer varias sustituciones seguidas, como en el caso de la composición secuencial. El orden de las sustituciones es importante y no hay que precipitarse, hay que hacerlas una a una para no equivocarse. Ejemplo, sea $P : x + y = N$,
$[[((P)^x_{x+y})^y_{3*x} \equiv ((x + y = N)^x_{x+y})^y_{3*x} \equiv (x + 2y = N)^y_{3*x} \equiv 7x = N]]$

Sobre deducciones lógicas: La notación $[[P]]$ indica que para todo estado σ, se cumple $P(\sigma)$, es decir se cumple $P(\sigma) \equiv cierto$. Por ejemplo, $\{P\}\, x := Exp\, \{Q\}$ se cumple si y sólo si $[[P \Rightarrow (Q)^x_{Exp}]]$. En otros textos se emplea otra notación para indicar lo mismo, por ejemplo $[[P \Rightarrow Q]]$ se escribe como $P \models Q$. En algunos contextos de demostración no utilizaremos los corchetes por quedar claro lo que se pretende. Donde no esté claro seguiremos utilizando la notación con corchetes.

Sobre algunas demostraciones: En bastantes ocasiones nos vamos a encontrar que tenemos que comprobar la certeza de $[[A \wedge B \wedge C \Rightarrow D \wedge E]]$ con distinto número de condiciones en el antecedente y en el consecuente de la implicación. Para hacer este tipo de demostraciones, es conveniente hacerlas por pasos, por ejemplo, comprobando la certeza de $[[A \wedge B \Rightarrow D]]$ y $[[C \Rightarrow E]]$ es suficiente para comprobar la proposición anterior.

Sobre algunos símbolos. Normalmente escribimos x: entero, el símbolo ':' lo leemos como 'es'; así, x es entero. Otro ejemplo, $x := x + y$, 'el valor de x es igual al valor que toma la expresión $x + y$. Cuando usamos este símbolo con predicados, por ejemplo P: $x \geq y$, decimos, 'el predicado P es el predicado $x \geq y$', y se entiende que $[[P \equiv x \geq y]]$ se cumple.

Sobre el lenguaje coloquial: Cuando tenemos un predicado P, esté será cierto o falso en un estado concreto σ. Hay que evaluar $P(\sigma)$ para saber si es cierto o no. Decir coloquialmente 'P se cumple' es una manera informal de decir 'sea σ un estado tal que $P(\sigma)$ es cierto' . También cuando un programa S satisface una especificación $\{P\} - \{Q\}$, decimos coloquialmente que 'S es correcto' pero hay que entender que es correcto con respecto a la especificación dada.

3. Pasos para la derivación de un programa

La estructura de un programa iterativo basado en un bucle para una especifiación $\{P\} - \{Q\}$ tiene la estructura siguiente (tal y como hemos indicado en la Lectura[4]):

1: **var**
2: variables ▷ declaración de variables del problema
3: **fvar**
4: $\{P\}$ ▷ precondición, condiciones iniciales
5: **var** variable auxiliares **fvar** ▷ variables de ayuda a la solución
6: S_{inicio} ▷ sentencias de inicio
7: $\{I\}$ ▷ Invariante del bucle
8: **mientras** B **hacer** ▷ B es la condición de continuación, $\neg B$ condición de salida
9: $\{I \wedge B\}$
10: S_{cuerpo} ▷ sentencias del cuerpo del bucle
11: $\{I\}$

12: **fmientras**
13: $\{Q\}$ ▷ postcondición, condiciones sobre los resultados

Las reglas de verificación para la estructura anterior presentadas en la Lectura[4] las ordenamos en la manera en la que debe realizarse el cálculo de las expresiones que se desconocen. Una guía elemental de la idea de la derivación de la solución para un problema dado se presenta a continuación. Se parte del conocimiento de la especificación $\{P\} - \{Q\}$ y de una propuesta de invariante I:

1. **Cálculo de la condición de salida y de continuación**. $[[\, I \wedge \neg B \Rightarrow Q \,]]$ se debe cumplir, es decir que $\{I \wedge \neg B\} \subseteq \{Q\}$. Conocemos la especificación $\{P\} - \{Q\}$ y la propuesta de invariante I, pero desconocemos la condición de salida $\neg B$. Se busca dicha condición y se calcula la condición de continuación B. Se comprueba que $[[\, I \Rightarrow def(B) \,]]$ es cierto, en el caso de que B contenga divisiones o expresiones que impliquen tablas. En otros casos $[[def(B)]]$ y no es necesaria la comprobación.

2. **Cálculo de las instrucciones de inicio**. $\{P\}\ S_{inicio}\ \{I\}$ se debe cumplir. Se buscan las sentencias más sencillas al principio de la ejecución para que se cumpla el invariante I. Las variables que aparecen en el invariante I que no tienen un valor inicial en P tienen que tomar un primer valor. Una vez propuestas estas sentencias hay que demostrar que $\{P\}$ $S_{inicio}\ \{I\}$ se cumple.

3. **Avanzar a la terminación**. $\{I \wedge B\}\ S_{cuerpo}\ \{I\}$ se debe cumplir. Como has calculado la condición de continuación en el paso 1, B, estudias cómo puedes hacer que se llegue a cumplir la condición de salida $\neg B$. Esa decisión te conduce a alguna sentencia de asignación que denominamos 'avanzar a la terminación'.

4. **Restablecer el invariante**. La sentencia o sentencias de terminación que has diseñado en el paso 3, dirige el diseño del resto del cuerpo del bucle. Imagina que esa sentencia es $x := E$, entonces escribes el axioma de la asignación $(I)^x_E$. Te resta de calcular las sentencias del cuerpo del bucle que cumplan $\{I \wedge B\}$ 'restablecer' $\{(I)^x_E\}$. Esas sentencias las denominamos 'sentencias de restablecer el invariante'.

5. **Programa y terminación**. Finalmente, si has diseñado todas las sentencias a partir de I y has demostrado la corrección, concluyes que I es invariante del bucle. Se escribe el programa correspondiente a la derivación anterior y se debe comprobar que termina, buscando una función de cota $f()$ con las dos propiedades siguientes:

   a) $[[\, I \wedge B \Rightarrow f() > 0 \,]]$
   b) $\{I \wedge B \wedge f() = V\}\, S_{cuerpo}\, \{f() < V\}$

No debes dar un nuevo paso hasta que hayas completado el anterior. En cualquier momento del proceso de derivavión puedes llegar a unas condiciones que te obliguen a revisar todo desde el principio. Al final, si has completado todos los pasos, el algoritmo obtenido es correcto. En ese caso, puedes estudiar su complejidad y pensar en otras maneras de resolver el problema.

4. Invariantes por elección de una conjunción

En algunos problemas, la postcondición Q es de una forma conjuntiva $Q : R \wedge L$. Una de esas dos conjunciones, R o L, puede indicar la razón por la que el programa termina. En ese caso elegimos aquella que indica la terminación, por ejemplo L, como condición de salida del bucle, $\neg B : L$, y la otra condición como invariante, $I : R$. Con esa elección, nos aseguramos que $[[I \wedge \neg B \Rightarrow Q]]$, puesto que $Q : R \wedge L$.

Ejemplo 1 *División entera de dos números.*

Consideremos la siguiente especificación.

1: **var**
2: x, y, q, r: entero
3: **fvar**
4: $\{P : x = A \wedge y = B \wedge x \geq 0 \wedge y > 0\}$
5: 'división entera'
6: $\{Q : q = A \text{ div } B \wedge r = A \text{ mod } B \wedge x = A \wedge y = B\}$

En la postcondición, q es el conciente de la división de A por B y r es el resto de la división de A por B. A y B son los valores iniciales de las variables x e y, y cumplen la condición $A \geq 0$ y $B > 0$. Estas variables no cambian su valor inicial. Al terminar, x e y siguen valiendo lo mismo que al principio. No tiene sentido hacer alguna instrucción de asignación sobre ellas en el programa. Por otro lado, A y B son valores dados al comienzo de la ejecución a estas variables y no son conocidos, ni están declarados (son variables de la especificación). No se pueden utilizar en las expresiones del programa. Cualquier programa que realicemos debe cumplir (por la interpretación operacional de las especificaciones) que 'para todos los valores de A y B, comenzando en un estado que cumple P, el programa termina; y termina en un estado que cumple Q'. Por ejemplo, para el estado inicial $\{x = 7 \wedge y = 2\}$, el programa debe terminar en el estado final $\{x = 7 \wedge y = 2 \wedge q = 3 \wedge r = 1\}$.

Por el teorema de la división entera (asumiendo $B > 0$) podemos reescribir la postcondición: $Q : A = qB + r \wedge 0 \leq r < B \wedge x = A \wedge y = B$, o de manera equivalente[2],

$Q : x = qy + r \wedge 0 \leq r < y \wedge x = A \wedge y = B$

(1) Propuesta de invariante y cálculo de la condición de salida. Recuerda que cuando haces la división a mano terminas cuando el resto es menor que el divisor, y eso mismo esta reflejado en la postcondición en la condición $r < y$. Elegimos esa condición como condición de salida del bucle $\neg B \equiv r < y$ y el resto como invariante:

$$I : x = qy + r \wedge 0 \leq r \wedge x = A \wedge y = B \wedge x \geq 0 \wedge y > 0$$
$$\neg B : r < y$$

Añadimos también las condiciones $x \geq 0 \wedge y > 0$ en la propuesta del invariante ya que la división es para números mayores o iguales a cero y requiere que el divisor sea distinto de cero. Con esa elección $[[\, I \wedge \neg B \Rightarrow Q \,]]$ se cumple directamente. Por tanto, la condición de continuación del bucle B es $r \geq y$.

(2) Cálculo de las instrucciones de inicio, $\{P\}\ S_{inicio}\ \{I\}$. Al principo $x = A$ y $y = B$. Hay que dar un valor inicial a q y a r. Los valores más simples para que se cumpla $x = qy + r$ son 0 y x. $\{P\}\ q := 0;\ r := x\ \{I\}$. Demostramos que $[[P \Rightarrow ((I)^r_x)^q_0]]$ (completa la prueba).

(3) Avanzar a la terminación. La condición de salida del bucle es $r < y$. La variable y no cambia su valor. La condición de continuación es $r \geq y$. Por tanto, $r - y \geq 0$. La variable r tiene que decrecer su valor, y una instrucción candidato para esto es $r := r - y$.

[2]Es conveniente que las variables de la especificación, como en este caso A y B, aparezcan el los predicados únicamente en igualdades con las variables que toman dicho valor y no en otras condiciones en el predicado.

(4) Restablecer el invariante. Hasta el momento tenemos el siguiente programa,

1: **var**
2: x, y, q, r: entero
3: **fvar**
4: $\{P : x = A \wedge y = B \wedge x \geq 0 \wedge y > 0\}$
5: $q := 0;$
6: $r := x;$
7: $\{I\}$
8: **mientras** $r \geq y$ **hacer**
9: $\{I \wedge B\}$
10: 'restablecer'
11: $\{(I)^r_{r-y}\}$
12: $r := r - y$
13: $\{I\}$
14: **fmientras**
15: $\{Q : x = qy + r \wedge 0 \leq r < y \wedge x = A \wedge y = B\}$

Nos queda calcular el conciente en la variable q. Escribimos los dos predicados siguientes:

$$I \wedge B : x = qy + r \wedge 0 \leq r \wedge x = A \wedge y = B \wedge x \geq 0 \wedge y > 0 \wedge r \geq y$$

$$(I)^r_{r-y} : x = qy + r - y \wedge 0 \leq r - y \wedge x = A \wedge y = B \wedge x \geq 0 \wedge y > 0$$

Asumiendo cualquier estado en el conjunto $\{I \wedge B\}$, todas las condiciones en un estado de $\{(I)^r_{r-y}\}$ se cumplen excepto $x = qy + r - y = (q-1)y + r$. La instrucción de 'restablecer' debe hacer que esto se cumpla a partir de $x = qy + r$. Observa que,
$x = qy + r \equiv (x = (q-1)y + r)^q_{q+1} \equiv x = (q + 1 - 1)y + r$, por lo que la instrucción de restablecer es $q := q + 1$. Podemos comprobar que, efectivamente, $[[I \wedge B \Rightarrow ((I)^r_{r-y})^q_{q+1}]]$ se cumple.

(5) Programa y terminación. El programa con el invariante I dado nos queda como,

1: **var**
2: x, y, q, r: entero
3: **fvar**
4: $\{P : x = A \wedge y = B \wedge x \geq 0 \wedge y > 0\}$
5: $q := 0;$
6: $r := x;$
7: $\{I : x = qy + r \wedge 0 \leq r \wedge x = A \wedge y = B \wedge x \geq 0 \wedge y > 0\}$ $//cota : r$
8: **mientras** $r \geq y$ **hacer**
9: $q := q + 1;$
10: $r := r - y$
11: **fmientras**
12: $\{Q : x = qy + r \wedge 0 \leq r < y \wedge x = A \wedge y = B\}$

La función de cota que garantiza que el bucle termina es $f() = r$, ya que (a) $[[I \wedge B \Rightarrow r \geq 0]]$, y (b) $(r)^r_{r-y} = r - y$ y $r - y < r$ puesto que $y > 0$ se cumple en I.

El valor inicial de q es 0, el valor final de q es A div B para los valores iniciales dados a las variables x e y. En cada vuelta del bucle, q se incrementa en una unidad. Por tanto, el número

de vueltas que da el bucle es A div B y su complejidad es $\mathcal{O}(A \text{ div } B)$. Se puede hacer un algoritmo para la división entera más eficiente que el dado, con coste $\mathcal{O}(\log(A \text{ div } B))$. □

Ejemplo 2 *Logaritmo en base dos por defecto de un número natural.*

Consideremos la siguiente especificación.

```
1: var
2:     x, k: entero
3: fvar
4: {P : x = X ∧ x ≥ 1}
5: 'logaritmo en base dos'
6: {Q : 2^k ≤ X < 2^(k+1)}
```

Dado un número natural X en la variable x, se debe calcular, en la variable k, el exponente que cumple $2^k \leq X < 2^{k+1}$. Por ejemplo, si x comienza con el valor 19, el resultado debe ser $k = 4$, puesto que $2^4 \leq 19 < 2^5$.

(1) Propuesta de invariante y cálculo de la condición de salida. La postcondición tiene dos conjunciones $Q : 2^k \leq X \wedge X < 2^{k+1}$. Elegimos una de ellas como condición de salida y la otra como invariante:

$$I : 2^k \leq X$$
$$\neg B : X < 2^{k+1}$$

Por tanto $[[I \wedge \neg B \equiv Q]]$. La condición de continuación B, según esto, es $X \geq 2^{k+1}$. La cuestión aquí es que no podemos programar X porque es una variable de la especificación. La **norma** es que sólo podemos acceder a los valores que contienen las variables a través de la variable declarada. No nos queda otro remedio que incluir en el invariante la condición $x = X$ y su condición $x \geq 1$. Por otra parte, la expresión 2^{k+1} no la podemos programar con el tipo entero. No tenemos directamente la potencia como operación básica del tipo. En este caso nos vemos obligados a **reforzar el invariante**. **Reforzar el invariante** es crear una nueva variable que, en el invariante, tome el valor de aquella expresión que no podemos programar. En este caso declaramos una variable auxiliar y, y añadimos al invariante la condición $y = 2^{k+1}$. Nos queda como propuesta de invariante y como condición de salida:

$$I : 2^k \leq x \wedge x = X \wedge x \geq 1 \wedge y = 2^{k+1}$$
$$\neg B : x < y$$

Observamos ahora que se cumple, $[[I \wedge \neg B \Rightarrow Q]]$:

$$I \wedge \neg B \equiv 2^k \leq x \wedge x = X \wedge x \geq 1 \wedge y = 2^{k+1} \wedge x < y$$
$$\equiv$$
$$2^k \leq X \wedge x = X \wedge x \geq 1 \wedge y = 2^{k+1} \wedge X < 2^{k+1}$$
$$\Rightarrow 2^k \leq X < 2^{k+1} \equiv Q$$

Por lo tanto, la condición de continuación B es $x \geq y$.

(2) Cálculo de las instrucciones de inicio. La variable x toma un valor natural X. El invariante elegido indica que en todo momento no cambia el valor de x (condición $x = X$ en I). No realizaremos instrucciones de asignación sobre está variable que puedan modificar su valor. Como $x \geq 1$, y en el invariante $2^k \leq x$, inicializamos la variable k a 0, lo que lleva a que al principio la variable y valga 2. Las instrucciones de inicio son, $k := 0$; $y :=2$. Debemos comprobar que

$[[P \Rightarrow ((I)_2^y)_0^k]]$ (completa la prueba).

(3) Avanzar a la terminación. La condición de continuación es $x \geq y$. x no cambia su valor, luego y debe crecer. En el invariante I, $y = 2^{k+1}$, Luego y crece si k crece. Observa que el resultado estará en la variable k. Decidimos avanzar con la instrucción $k := k + 1$.

(4) Restablecer el invariante. Hasta el momento tenemos el siguiente programa,

```
1: var
2:     x, k: entero
3: fvar
4: {P : x = X ∧ x ≥ 1}
5: var y: entero fvar
6: k := 0;
7: y := 2;
8: {I}
9: mientras x ≥ y hacer
10:    {I ∧ B}
11:    'restablecer'
12:    {(I)^k_{k+1}}
13:    k := k + 1
14:    {I}
15: fmientras
16: {Q : 2^k ≤ X < 2^{k+1}}
```

Escribimos los dos predicados siguientes,

$$I \wedge B : v2^k \leq x \wedge x = X \wedge x \geq 1 \wedge y = 2^{k+1} \wedge x \geq y$$

$$(I)_{k+1}^k : 2^{k+1} \leq x \wedge x = X \wedge x \geq 1 \wedge y = 2^{k+2}$$

Todas las condiciones en un estado de $\{I \wedge B\}$ se cumplen en un estado de $\{(I)_{k+1}^k\}$ excepto que en este último $y = 2^{k+2}$. El valor anterior de y es 2^{k+1} y el nuevo valor que debe tomar y es 2^{k+2}. La variable y cambia su valor, luego la instrucción de restablecer es una instrucción de asignación sobre y. Asumiendo cualquier estado del conjunto $\{I \wedge B\}$, entonces,

$$\begin{aligned} & 2^{k+2} \\ &= 2\,2^{k+1} \text{ supuesto } \{I \wedge B\} \\ &= 2y \end{aligned}$$

Por lo que la instrucción de restablecer[3] es $y := 2 * y$.

(5) Programa y terminación. El programa que resulta de la derivación anterior es el siguiente,

```
1: var
2:     x, k: entero
3: fvar
4: {P : x = X ∧ x ≥ 1}
5: var y: entero fvar
```

[3]Este proceso es muy general. No importa lo complicados que sean los predicados. Cuando una variable debe cambiar su valor entre dos estados en la ejecución, tienes que intentar obtener el nuevo valor a partir de los valores contenidos en las variables en el estado anterior.

6: $k := 0$;
7: $y := 2$;
8: $\{I : 2^k \leq x \wedge x = X \wedge x \geq 1 \wedge y = 2^{k+1}\}$ // *cota* : $x - k$
9: **mientras** $x \geq y$ **hacer**
10: $y := 2 * y$;
11: $k := k + 1$
12: **fmientras**
13: $\{Q : 2^k \leq X < 2^{k+1}\}$

La función de cota que garantiza que el bucle termina es $f() = x - k$, ya que,
(a) $[[I \wedge B \Rightarrow x - k > 0]]$,

$$
\begin{array}{l}
2^k \leq x \wedge x = X \wedge x \geq 1 \wedge y = 2^{k+1} \wedge x \geq y \\
\Rightarrow \\
x \geq 2^{k+1} \\
\Rightarrow (\text{ por } 2^{k+1} > k) \\
x > k \equiv x - k > 0
\end{array}
$$

(b) En cada paso del bucle la única instrucción que afecta a la funcion $f() = x - k$, es $k := k + 1$, por tanto, decrece, ya que $(x - k)^k_{k+1} = x - k - 1 < x - k$.

El número de vueltas que da el bucle es proporcional a $\log_2(X)$. □

5. Invariantes por sustitución de constantes por variables

Una gran cantidad de problemas algorítmicos tratan sobre el cálculo de alguna expresión con cuantificadores, por ejemplo: sumar los elementos de una tabla, contar el número de veces que en una tabla hay elementos positivos, sumar los divisores de un número, calcular el número pi por el producto de Wallis, calcular el producto de dos matrices, realizar el producto escalar de dos vectores. En el planteamiento de estos problemas, en general, no hay una condición de salida que permita obtener el invariante directamente de la postcondición como hemos visto en la sección anterior, más bien, hay que construir esa condición de salida. El método que se emplea en estos casos es el de sustitución de constantes por variables. Para explicar el método, comenzaremos con el ejemplo de sumar los elementos de una tabla.

Ejemplo 3 *Suma de los elementos de un vector de enteros.*

Consideremos la siguiente especificación:

1: **constante**
2: N $(N \geq 1)$
3: **fconstante**
4: **tipo**
5: vector: **tabla** $[0..N-1]$ **de** entero
6: **ftipo**
7: **var**
8: t: vector;
9: s: entero
10: **fvar**
11: $\{P : t = T\}$
12: 'suma de los elementos de un vector'
13: $\{Q : s = (\sum \gamma : 0 \leq \gamma < N : t[\gamma]) \wedge t = T\}$

Consideremos un ejemplo. Sea $N = 4$, las posiciones del vector (según la declaración del tipo) van de 0 hasta 3. Sea al comienzo t, el vector $(-2, 3, -1, 7)$. La variable s al final de cualquier ejecución debe valer $s = t[0] + t[1] + t[2] + t[3]$. Ya que el vector t no cambia sus valores desde el principio, en el ejemplo dado s es 7. El rango del cuantificador $0 \leq \gamma < N$, asegura que los elementos a los que se accede van en las posiciones desde 0 hasta $N - 1$. La idea es que para realizar el cálculo anterior necesitamos una variable que haga el recorrido por la posiciones desde 0 hasta $N - 1$ y que cuando llegue a valer N el cálculo termine. En todo momento antes de sumar un nuevo elemento la variable s contiene la suma hasta la última posición antes de k, y esto nos va a conducir al invariante en el que la constante N del cuantificador se sustituye por la variable k:

$$I : s = (\sum \gamma : 0 \leq \gamma < k : t[\gamma]) \wedge 0 \leq k \leq N \wedge t = T$$

Para obtener la propuesta de invariante I, hemos sustituido en Q la constante N por una nueva variable k; como k sustituye a N no será mayor que N, y como la primera posición en la tabla es 0, entonces $0 \leq k \leq N$. Esta condición se incluye en la propuesta del invariante.

Nota: al ser N una constate, su condición $N \geq 1$ se da por cierta en toda la derivación.

(1) Cálculo de la condición de salida. Por la propia forma de construcción de I, cuando $k = N$, se cumple la postcondición Q, es decir, $[[I \wedge k = N \Rightarrow Q]]$,

$$\begin{array}{l} I \wedge k = N \\ \equiv \\ s = (\sum \gamma : 0 \leq \gamma < k : t[\gamma]) \wedge 0 \leq k \leq N \wedge t = T \wedge k = N \\ \equiv \\ s = (\sum \gamma : 0 \leq \gamma < N : t[\gamma]) \wedge 0 \leq N \wedge t = T \wedge k = N \\ \Rightarrow \\ s = (\sum \gamma : 0 \leq \gamma < N : t[\gamma]) \wedge t = T \end{array}$$

Por lo tanto, la condición de continuación B es $k \neq N$.

(2) Cálculo de las instrucciones de inicio. La suma sobre un rango vacío es cero. Como $0 \leq k \leq N$ en I, y el programa termina en $k = N$, probamos el inicio con k valiendo 0, en ese caso el rango del cuantificador se hace vacío y la variable s debe valer 0 también. Luego proponemos como instrucciones de inicio $k := 0$; $s := 0$. Comprobamos que $[[P \Rightarrow ((I)_0^s)_0^k]]$,

$$\begin{array}{l} ((I)_0^s)_0^k \\ \equiv \\ 0 = (\sum \gamma : 0 \leq \gamma < 0 : t[\gamma]) \wedge 0 \leq 0 \leq N \wedge t = T \\ \equiv \text{ (rango vacío)} \\ 0 = 0 \wedge 0 \leq N \wedge t = T \\ \Leftarrow \text{ (constante } N \geq 1) \\ t = T \end{array}$$

(3) Avanzar a la terminación. La condición de salida es $k = N$. La variable k al inicio comienza en 0 y debe terminar en N. Se incrementa de uno en uno por la lógica de la suma. Proponemos avanzar con $k := k + 1$.

(4) Restablecer el invariante. Hasta el momento tenemos lo siguiente,

1: **constante**
2: N $(N \geq 1)$
3: **fconstante**

```
 4: tipo
 5:     vector: tabla [0..N − 1] de entero
 6: ftipo
 7: var
 8:     t: vector;
 9:     s: entero
10: fvar
11: {P : t = T}
12: var k : entero fvar
13: k := 0;
14: s := 0;
15: {I}
16: mientras k ≠ N hacer
17:     {I ∧ B}
18:     'restablecer'
19:     {(I)^k_{k+1}}
20:     k := k + 1
21:     {I}
22: fmientras
23: {Q : s = (∑γ : 0 ≤ γ < N : t[γ]) ∧ t = T}
```

Escribimos los predicados,

$$I \wedge B :\ s = (\sum \gamma : 0 \leq \gamma < k : t[\gamma]) \wedge 0 \leq k \leq N \wedge t = T \wedge k \neq N$$

$$(I)^k_{k+1} :\ s = (\sum \gamma : 0 \leq \gamma < k+1 : t[\gamma]) \wedge 0 \leq k+1 \leq N \wedge t = T$$

Las condiciones $0 \leq k \leq N \wedge k \neq N$, establecen que $[[0 \leq k < N \Rightarrow 0 \leq k+1 \leq N]]$. El valor de s cambia y se requiere una instrucción de asignación,

$(\sum \gamma : 0 \leq \gamma < k+1 : t[\gamma])$ (el nuevo valor de s)
$=$ (separando por el último término)
$(\sum \gamma : 0 \leq \gamma < k : t[\gamma]) + t[k]$
$=$ (asumiendo $\{I \wedge B\}$, por, $0 \leq k < N$, y el valor de s)
$s + t[k]$

Por lo que la instrucción de asignación buscada es $s := s + t[k]$.

(5) Programa y terminación. El programa que resulta de la derivación anterior es el siguiente,

```
 1: constante
 2:     N (N ≥ 1)
 3: fconstante
 4: tipo
 5:     vector: tabla [0..N − 1] de entero
 6: ftipo
 7: var
 8:     t: vector;
 9:     s: entero
10: fvar
11: {P : t = T}
12: var k : entero fvar
```

13: $k := 0$;
14: $s := 0$;
15: $\{I : s = (\sum \gamma : 0 \leq \gamma < k : t[\gamma]) \wedge 0 \leq k \leq N \wedge t = T\}$ // *cota* : $N - k$
16: **mientras** $k \neq N$ **hacer**
17: $\quad s := s + t[k]$;
18: $\quad k := k + 1$;
19: **fmientras**
20: $\{Q : s = (\sum \gamma : 0 \leq \gamma < N : t[\gamma]) \wedge t = T\}$

La función de cota que garantiza que el bucle termina es $f() = N - k$, ya que, (a) $[[I \wedge B \Rightarrow N - k > 0]]$; y (b), la única instrucción en el cuerpo del bucle que afecta a dicha función de cota, es $k := k + 1$, luego $(N - k)^k_{k+1} = N - k - 1$ y $N - k - 1 < N - k$, entonces la función de cota decrece en cada vuelta del bucle.

El bucle da N vueltas luego la complejidad es $\mathcal{O}(N)$. No se puede hacer más rápido porque tienes que acceder a los N elementos para conocer su valor. Obligatoriamente tienes N accesos a la tabla y cada acceso cuenta como una operación. □

Ejemplo 4 *Número de elementos pares en un vector.*

Consideremos la siguiente especificación:

1: **constante**
2: $\quad N$ $(N \geq 1)$
3: **fconstante**
4: **tipo**
5: $\quad$ vector: **tabla** $[0..N - 1]$ **de** entero
6: **ftipo**
7: **var**
8: $\quad$ t: vector;
9: $\quad$ r: entero
10: **fvar**
11: $\{P : t = T\}$
12: 'suma de los elementos de un vector'
13: $\{Q : r = (\#\gamma : 0 \leq \gamma < N : t[\gamma] \bmod 2 = 0) \wedge t = T\}$

El resultado en la variable r indica el número de elementos de la tabla cuyo valor es un número par. Por ejemplo, si al comienzo la tabla t es $(2, 3, -1, 8)$ el resultado en r es 2. En este ejemplo se usa el mismo tipo vector y posiciones que se ha utilizado en el ejemplo anterior. El programa se deriva de forma similar al ejemplo de la suma. El invariante se construye de nuevo sustituyendo en la postcondición Q la constante N por una nueva variable k, de manera que la propuesta de invariante y la condición de salida son,

$$I : r = (\#\gamma : 0 \leq \gamma < k : t[\gamma] \bmod 2 = 0) \wedge 0 \leq k \leq N \wedge t = T$$
$$\neg B : k = N$$

El proceso de derivación de las instrucciones de inicio y la instrucción de avanzar es muy similar al dado en el Ejemplo 3 anterior, por lo que sólo calcularemos la sentencia de restablecer el invariante. La derivación nos lleva a lo siguiente,

1: **constante**
2: $\quad N$ $(N \geq 1)$
3: **fconstante**

4: **tipo**
5: vector: **tabla** $[0..N-1]$ **de** entero
6: **ftipo**
7: **var**
8: t: vector;
9: r: entero
10: **fvar**
11: $\{P : t = T\}$
12: **var** k : entero **fvar**
13: $k := 0$;
14: $r := 0$;
15: $\{I\}$
16: **mientras** $k \neq N$ **hacer**
17: $\{I \wedge B\}$
18: 'restablecer'
19: $\{(I)^k_{k+1}\}$
20: $k := k + 1$
21: $\{I\}$
22: **fmientras**
23: $\{Q : r = (\#\gamma : 0 \leq \gamma < N : t[\gamma] \bmod 2 = 0) \wedge t = T\}$

Escribimos los predicados,

$$I \wedge B : \; r = (\#\gamma : 0 \leq \gamma < k : t[\gamma] \bmod 2 = 0) \wedge 0 \leq k \leq N \wedge t = T \wedge k \neq N$$

$$(I)^k_{k+1} : \; r = (\#\gamma : 0 \leq \gamma < k+1 : t[\gamma] \bmod 2 = 0) \wedge 0 \leq k+1 \leq N \wedge t = T$$

Las condiciones $0 \leq k \leq N \wedge k \neq N$, establecen que $[[0 \leq k < N \Rightarrow 0 \leq k+1 \leq N]]$. El valor de la variable r cambia entre los dos predicados. Estudiamos la relación entre el nuevo valor que debe tomar la variable r con respecto al valor que tiene en el estado anterior.

$(\#\gamma : 0 \leq \gamma < k+1 : t[\gamma] \bmod 2 = 0)$ (el nuevo valor de r)
= (separar por el último término)
$(\#\gamma : 0 \leq \gamma < k : t[\gamma] \bmod 2 = 0) + \#(t[k] \bmod 2 = 0)$
= (asumiendo $\{I \wedge B\}$, por, $0 \leq k < N$, y el valor de r)

(caso 1.) $= r + 1$ si $t[k] \bmod 2 = 0$ se cumple
(caso 2.) $= r$ si $t[k] \bmod 2 \neq 0$ se cumple

Según lo anterior, tenemos que los siguientes fragmentos son correctos,

$$\{I \wedge B \wedge t[k] \bmod 2 = 0\}$$
$$r := r + 1$$
$$\{(I)^k_{k+1}\}$$

$$\{I \wedge B \wedge t[k] \bmod 2 \neq 0\}$$
$$\text{continuar}$$
$$\{(I)^k_{k+1}\}$$

La sentencia de restablecer es ahora una alternativa exclusiva con dos casos. El programa nos queda entonces como sigue:

```
1: constante
2:     N (N ≥ 1)
3: fconstante
4: tipo
5:     vector: tabla [0..N − 1] de entero
6: ftipo
7: var
8:     t: vector;
9:     r: entero
10: fvar
11: {P : t = T}
12: var k : entero fvar
13: k := 0;
14: r := 0;
15: {I : r = (#γ : 0 ≤ γ < k : t[γ] mod 2 = 0) ∧ 0 ≤ k ≤ N ∧ t = T} // cota : N − k
16: mientras k ≠ N hacer
17:     si
18:     [] t[k] mod 2 = 0 →
19:                                 r := r + 1
20:     [] t[k] mod 2 ≠ 0 →
21:                                 continuar
22:     fsi;
23:     k := k + 1
24: fmientras
25: {Q : r = (#γ : 0 ≤ γ < N : t[γ] mod 2 = 0) ∧ t = T}
```

□

Ejemplo 5 *Posición del valor máximo en una tabla.*

En el siguiente ejemplo tratamos con una tabla de N elementos enteros y buscamos la posición que ocupa el elemento con el valor máximo. Los índices de la tabla van desde 1 hasta N. La especificación es la siguiente,

```
constante
    N (N ≥ 1)
fconstante
tipo
    tabla1N: tabla [1..N] de entero
ftipo
var
    t: tabla1N;
    x: entero
fvar
{P : t = T}
'posición del máximo'
{Q : t[x] = (max γ : 1 ≤ γ ≤ N : t[γ]) ∧ t = T}
```

La operación max entre dos números, a max b, nos devuelve el mayor entre ellos. Es una operación conmutativa. También es una operación asociativa, ya que $(a \text{ max } b) \text{ max } c = a \text{ max } (b \text{ max } c)$. No tiene elemento neutro; en su caso sería $-\infty$, $a \text{ max } -\infty = a$. La operación max admite un

cuantificador como en el caso que nos ocupa. En la postcondición $(\max\ \gamma : 1 \leq \gamma \leq N : t[\gamma])$ devuelve el valor mayor entre todos los elementos de la tabla, pero observa que en Q, $t[x] = (\max\ \gamma : 1 \leq \gamma \leq N : t[\gamma])$, es decir, dicho valor se encuentra en el elemento que ocupa la posición x, y es justamente x el resultado que queremos. Así, por ejemplo, si la tabla t fuese $(2, 3, -1, 8, -3, 6)$ al principio de la ejecución (para $N = 6$) el resultado debe ser $x = 4$, ya que en la posición 4, encontramos que $t[4]$ contiene el mayor valor de todos. El problema planteado es un problema de 'recorrido' ya que debes mirar todos y cada uno de los elementos para buscar cuál es el mayor y su posición. El método para construir el invariante es de nuevo 'sustitución de constantes por variables'. Sustituimos N en Q por una variable k.

$$I : t[x] = (\max\ \gamma : 1 \leq \gamma \leq k : t[\gamma]) \wedge 1 \leq k \leq N \wedge t = T$$

En la propuesta de invariante, k sustituye a N; entonces no puede ser mayor que N y como la primera posición es 1, debe cumplir $1 \leq k \leq N$. Observa también que no vamos a permitir que el rango del cuantificador se haga vacío puesto que el máximo no tiene elemento neutro que puedas programar (no tienes un número entero que sea $-\infty$). La derivación del programa sigue los mismos pasos dados en los ejemplos anteriores.

(1) Cálculo de la condición de salida. Como $[[I \wedge \neg B \Rightarrow Q]]$ debe cumplirse, es simple ver que la condición de salida $\neg B$ es $k = N$. Con lo que la condición de continuación B es $k \neq N$.

(2) Cálculo de las instrucciones de inicio. Si el bucle termina cuando $k = N$, y el rango de k en el invariante es $1 \leq k \leq N$ podemos probar en iniciar la variable k a valor 1. En ese caso,

$$\begin{array}{l}(\max\ \gamma : 1 \leq \gamma \leq k : t[\gamma]) \text{ cuando } k = 1\\ =\\ (\max\ \gamma : 1 \leq \gamma \leq 1 : t[\gamma])\\ =\\ t[1]\end{array}$$

Por tanto, como $t[x]$ debe ser $t[1]$ (el máximo hasta ese punto), x debe ser 1 también. Las instrucciones de inicio serán $k := 1$; $x := 1$;. Puedes comprobar entonces que $[[P \Rightarrow ((I)_1^x)_1^k]]$ (completa la prueba).

(3) Avanzar a la terminación. La variable k comienza en 1 y debe terminar con valor N por lo que proponemos avanzar con $k := k + 1$. Además, hay que 'visitar' a todos los elementos de la tabla uno a uno.

(4) Cálculo de la sentencia de restablecer. Llegados a este punto tenemos el siguiente programa,

constante
  N $(N \geq 1)$
fconstante
tipo
  tabla1N: **tabla** $[1..N]$ **de** entero
ftipo
var
  t: tabla1N;
  x: entero
fvar
$\{P : t = T\}$
var k : entero **fvar**

$k := 1;$
$x := 1;$
$\{I\}$
mientras $k \neq N$ **hacer**
  $\{I \wedge B\}$
  'restablecer'
  $\{(I)^k_{k+1}\}$
  $k := k + 1$
  $\{I\}$
fmientras
$\{Q : t[x] = (\max\ \gamma : 1 \leq \gamma \leq N : t[\gamma]) \wedge t = T\}$

Escribimos los dos predicados que nos peritirán diseñar el restablecer,

$$I \wedge B : t[x] = (\max\ \gamma : 1 \leq \gamma \leq k : t[\gamma]) \wedge 1 \leq k \leq N \wedge t = T \wedge k \neq N$$

$$(I)^k_{k+1} : t[x] = (\max\ \gamma : 1 \leq \gamma \leq k+1 : t[\gamma]) \wedge 1 \leq k+1 \leq N \wedge t = T$$

Como $[[1 \leq k \leq N \wedge k \neq N \Rightarrow 1 \leq k+1 \leq N]]$ se cumple, nos resta ver que $t[x]$ pueden ser 'distintos' en ambos predicados. Estudiamos que relación hay entre $(\max\ \gamma : 1 \leq \gamma \leq k+1 : t[\gamma])$ y $(\max\ \gamma : 1 \leq \gamma \leq k : t[\gamma])$ en las condiciones de un estado que cumpla $I \wedge B$,

$(\max\ \gamma : 1 \leq \gamma \leq k+1 : t[\gamma])$
$=$ (separando por el último término)
$(\max\ \gamma : 1 \leq \gamma \leq k : t[\gamma])\ \max\ t[k+1]$
$=$ (asumiendo $\{I \wedge B\},$ por, $1 \leq k < N,$ y el valor de t[x])
$t[x]\ \max\ t[k+1]$
$=$ (por la definición de max)
(caso 1.) $= t[x]$ si $t[x] \geq t[k+1]$ se cumple
(caso 2.) $= t[k+1]$ si $t[x] < t[k+1]$ se cumple

En el primer caso, si $t[x] \geq t[k+1]$, entonces el nuevo valor de $t[x]$ es el mismo que el anterior, luego la instrucción será 'continuar'. En el segundo caso, si $t[x] < t[k+1]$, el nuevo valor de $t[x]$ debe ser $t[k+1]$ y para consegir esto hay que 'actualizar' el valor de x a la nueva posición que ocupa el máximo, es decir, $x := k + 1$. Como los dos casos son exclusivos se requiere una composición alternativa.

(5) Programa y terminación. El programa que resulta de la derivación anterior es el siguiente,

constante
  N $(N \geq 1)$
fconstante
tipo
  tabla1N: **tabla** $[1..N]$ **de** entero
ftipo
var
  t: tabla1N;
  x: entero
fvar
$\{P : t = T\}$
var k : entero **fvar**
$k := 1;$

$x := 1$;
$\{I : t[x] = (\max\ \gamma : 1 \leq \gamma \leq k : t[\gamma]) \wedge 1 \leq k \leq N \wedge t = T\}$ // $cota : N - k$
mientras $k \neq N$ **hacer**
  si
  [] $t[x] \geq t[k+1] \rightarrow$ continuar
  [] $t[x] < t[k+1] \rightarrow x := k + 1$
  fsi;
  $k := k + 1$
fmientras
$\{Q : t[x] = (\max\ \gamma : 1 \leq \gamma \leq N : t[\gamma]) \wedge t = T\}$

La función de cota que garantiza que el bucle termina es $f() = N - k$, ya que, (a) $[[I \wedge B \Rightarrow N - k > 0]]$ y; (b), la única instrucción en el cuerpo del bucle que afecta a dicha función de cota es $k := k + 1$, luego $(N - k)^k_{k+1} = N - k - 1$ y $N - k - 1 < N - k$, entonces la función de cota decrece en cada vuelta del bucle. Finalmente, el bucle da N vueltas luego su complejidad es $\mathcal{O}(N)$. □

Nota al programador: Como habrás observado en los ejemplos anteriores, puedes diseñar algoritmos para tamaños de tablas iguales pero conjunto de índices distintos. En los tres primeros ejemplos el tamaño de la tabla era N y los índices iban desde 0 hasta $N - 1$. En el último ejemplo, el tamaño de la tabla era N pero los índices iban desde 1 hasta N. En los lenguajes de programación, las tablas ('arrays') tienen sus particularidades: hay lenguajes diferentes y formas diferentes para los índices. Por ejemplo, en el Lenguaje C, que vas a utilizar en prácticas, declaras el tamaño de la tabla N pero sus índices quedan fijados desde 0 hasta $N-1$. En el entorno de Matlab los índices van de 1 a N. En Pascal, puedes declarar los índices como quieras utilizando un tipo básico totalmente ordenado. Cuando hacemos los diseños de programas con tablas en nuestro pseudocódigo nos preocupamos mucho de que los accesos a la tabla sean legales, es decir, no permitimos acceder a una posición de la tabla que no está declarada, y la derivación y la verificación tiene en cuenta siempre este hecho. El lenguaje Pascal y Matlab, por ejemplo, si accedes a una posición no declarada en la tabla, la ejecución se termina dando un error. En cambio en C se puede seguir ejecutando accediendo a posiciones de memoria que no se corresponden con la delcaración, y esto, puede suponer un funcionamiento del programa totalmente alejado de lo que se pretendía. Para hacer que los algoritmos diseñados sean independientes del lenguaje de programación que luego se va a emplear para su codificación es conveniente declarar dos constantes L y U ($L \leq U$), tales que determinen las posiciones $[L..U]$ que admite la tabla. El tamaño de la tabla queda determinado por dichas posiciones $U - L + 1$. Esta misma idea se puede extender a tablas con varias dimensiones. Por ejemplo, la especificación del ejemplo del máximo quedaría como,

constante
  L ▷ primera posición
  U, $L \leq U$ ▷ última posición. Tamaño de tabla $U - L + 1$
fconstante
tipo
  tablaLU: **tabla** $[L..U]$ **de** entero
ftipo
var
  t: tabla1N;
  x: entero
fvar
$\{P : t = T\}$
'posición del máximo'
$\{Q : t[x] = (\max\ \gamma : L \leq \gamma \leq U : t[\gamma]) \wedge t = T\}$

6. Reforzamiento de invariantes

Como hemos visto en los ejemplos anteriores, la derivación permite obtener algoritmos correctos a partir de una propuesta de invariante. Lo que dirige el diseño es la elección de la condición de salida y la elección de la instrucción o instrucciones que avanzan a la terminación del bucle. En el caso de que la condición de continuación no pueda programarse debido a que contiene expresiones con operaciones que no están definidas en sus tipos, es obligatorio reforzar el invariante con una nueva variable que es igual a la expresión que no se puede programar. En otros casos, cuando se trata de restablecer el invariante, también pueden aparecer expresiones que no son programables. En estos casos tiene sentido reforzar el invariante siempre que eso suponga una ventaja a la hora de diseñar el programa [4]. En lo que sigue mostraremos dos ejemplos de lo que estamos indicando en este párrafo. El primer ejemplo es bastante simple: el cálculo del término N-ésimo de la sucesión de Fibonnaci. El segundo ejemplo con una tabla es un poco más complicado, pero es ilustrativo de la importancia de la derivación.

Ejemplo 6 *Término N-ésimo de la sucesión de Fibonacci.*

La sucesión de Fibonacci, fib(), se define como,

$$\text{fib}(0) = 0,\ \text{fib}(1) = 1$$
$$\text{fib}(n+2) = \text{fib}(n) + \text{fib}(n+1) \text{ para } n \geq 0$$

Se pide realizar un programa para calcular fib(N), según la siguiente especificación,

```
1: var
2:     n, f: entero
3: fvar
4: {P : n = N ∧ n ≥ 0}
5: 'Fibonacci'
6: {Q : f = fib(n) ∧ n = N}
```

La variable n no cambia su valor. Indica el término de la sucesión que hay que calcular. Por ejemplo, si n comienza con valor 6, n termina con valor 6 y f contiene el valor fib(6), que es 8 (0, 1, 1, 2, 3, 5, 8,...). Como n, aún siendo una variable, actua como una constante (no vamos a hacer ninguna instrucción de asiganción sobre ella), proponemos un invariante por sustitución de n por una nueva variable i en la expresión $f = \text{fib}(n)$ y mantenemos el hecho de que $n = N$. La propuesta de invariante es,

$$I : f = \text{fib}(i) \wedge n = N \wedge 0 \leq i \leq n$$

Observa la condición, $0 \leq i \leq n$, que indica que i, al sustituir a n no será más grande que n, y como n es ≥ 0, lo mismo sucede con i.

(1) Condición de salida. La condición $i = n$ conduce a que $[[I \wedge i = N \Rightarrow Q]]$. Luego la condición de continuación B es $i \neq n$.

(2) Instrucciones de inicio. Al principio $i := 0$; $f := 0$ establece el invariante, ya que $[[P \Rightarrow ((I)_0^f)_0^i]]$ se cumple.

(3) Avanzar a la terminación. Obviamente $i := i + 1$.

[4] En el caso de que el reforzamiento no sea beneficioso entonces deberemos diseñar un nuevo programa para esa expresión dentro del programa que estamos diseñando.

(4) Restablecer el invariante. Escribimos los dos predicados siguientes,

$$I \wedge B : f = \text{fib}(i) \wedge n = N \wedge 0 \leq i \leq n \wedge i \neq n$$

$$(I)^i_{i+1} : f = \text{fib}(i+1) \wedge n = N \wedge 0 \leq i+1 \leq n$$

La expresión $\text{fib}(i+1)$ no puede programarse únicamente con los valores de i y f anteriores. Luego reforzamos el invariante con una nueva variable g igual a la expresión que no podemos programar $\text{fib}(i+1)$. La nueva propuesta de invariante queda como sigue,

$$I' : f = \text{fib}(i) \wedge n = N \wedge 0 \leq i \leq n \wedge g = \text{fib}(i+1)$$

Se debe repasar de nuevo la derivación. (1) La condición de salida no cambia. (2) Se debe añadir una nueva instrucción $g{:=}\ 1$ para que $[[P \Rightarrow (((I')^g_1)^f_0)^i_0]]$ se cumpla. (3) No cambia la instrucción de avanzar a la terminación. Queda por tanto restablecer el invariante,

(4) Restablecer el invariante. Escribimos los dos predicados siguientes,

$$I' \wedge B : f = \text{fib}(i) \wedge n = N \wedge 0 \leq i \leq n \wedge g = \text{fib}(i+1) \wedge i \neq n$$

$$(I')^i_{i+1} : f = \text{fib}(i+1) \wedge n = N \wedge 0 \leq i+1 \leq n \wedge g = \text{fib}(i+2)$$

Asumiendo que estamos en un estado que cumple $I' \wedge B$, entonces

$$\begin{array}{l} \text{fib}(i+1) \text{ (el nuevo valor de f)} \\ = g \text{ (el valor anterior de g)} \end{array}$$

$$\begin{array}{l} \text{fib}(i+2) \text{ (el nuevo valor de g)} \\ = \text{fib}(i) + \text{fib}(i+1) \text{ (por definición de fib())} \end{array}$$

$$= f + g \text{ (los valores anteriores de f y g)}$$

Observa también que $[[0 \leq i < n \Rightarrow 0 \leq i+1 \leq n]]$. Por lo tanto, según lo anterior la asignación mútiple $\langle f, g\rangle := \langle g, f+g\rangle$ restablece el invariante. El programa que resulta de la derivación anterior es,

```
1: var
2:     n, f: entero
3: fvar
4: {P : n = N ∧ n ≥ 0}
5: var g : entero fvar
6: i := 0; f := 0; g := 1;
7: {I' : f = fib(i) ∧ n = N ∧ 0 ≤ i ≤ n ∧ g = fib(i+1)} // cota : n − i
8: mientras i ≠ n hacer
9:     ⟨f, g⟩ := ⟨g, f + g⟩;
10:    i := i + 1
11: fmientras
12: {Q : f = fib(n) ∧ n = N}
```

La función de cota que asegura la terminación del programa anterior es $f() = n - i$. La complejidad de este algoritmo es $\mathcal{O}(N)$. Hay que hacer un cambio un poco radical de la manera de pensar en la solución para obtener un algoritmo con una complejidad de tipo $\mathcal{O}(\log(N))$. Una

última cuestión que debes hacer es cambiar la instrucción de asignación múltiple por asignaciones simples. □

Ejemplo 7 *Cuenta parejas de diferente signo.*

Considera una tabla t con índices de 1 hasta N siendo $N \geq 1$ una constante. El problema trata de contar las parejas (i, j) con $1 \leq i < j \leq N$ tales que $t[i] < 0$ y $t[j] > 0$. Por ejemplo, para $N = 5$ y una tabla t, $(-1, 2, -3, 0, 4)$, el número de dichas parejas sería 3, ya que $(1, 2)$, $(1, 5)$, y $(3, 5)$ cumplen la condición, el resto no la cumple. Formalmente la especificación es,

constante
  N $(N \geq 1)$
fconstante
tipo
  tabla1N: **tabla** $[1..N]$ **de** entero
ftipo
var
  t: tabla1N;
  r: entero
fvar
$\{P : t = T\}$
'cuenta parejas de diferente signo'
$\{Q : r = (\#i, j : 1 \leq i < j \leq N : t[i] < 0 \wedge t[j] > 0) \wedge t = T\}$

Observa que el cuantificador tiene dos variables ligadas i y j (por mayor claridad no las he escrito con letras griegas). Dado que la postcondición incluye un cuantificador de recuento, proponemos el invariante sustituyendo la constante N por una nueva variable n. Así,

$$I : r = (\#i, j : 1 \leq i < j \leq n : t[i] < 0 \wedge t[j] > 0) \wedge t = T \wedge 1 \leq n \leq N$$

Para mayor comodidad en el desarrollo, como la tabla no cambia sus valores asumimos que $t = T$, y no lo volvemos a escribir en los predicados. De forma similar asumimos invariante la condición de la constante $N \geq 1$. El invariante lo redefinimos para luego poder añadir nuevas condiciones si se necesita.

$$I_1 : r = (\#i, j : 1 \leq i < j \leq n : t[i] < 0 \wedge t[j] > 0)$$
$$I_2 : 1 \leq n \leq N$$
$$I_1 \wedge I_2 \text{ propuesta inicial de invariante}$$

(1) Cálculo de la condición de salida. Por la propia construcción del invariante, cuando $n = N$, se cumple que $[[I_1 \wedge I_2 \wedge n = N \Rightarrow Q]]$. La condición de continuación es $B : n \neq N$.

(2) Instrucciones de inicio. Observa que cuando n es 1 el rango del cuantificador es vacío y por definición de dicho cuantificador vale 0. Luego las instrucciones de inicio serán $n := 1$; $r := 0$. Comprueba que $[[P \Rightarrow ((I_1 \wedge I_2)_0^r)_1^n]]$.

(3) Avanzar a la terminación. La variable n comienza inicialmente en 1 y debe terminar en N, luego avanzamos con $n := n + 1$.

(4) Restablecer el invariante. Hasta el momento el programa (anotado con los conjuntos de estados) es el siguiente,

constante
N ($N \geq 1$)
fconstante
tipo
tabla1N: **tabla** [1..N] **de** entero
ftipo
var
t: tabla1N;
r: entero
fvar
$\{P : t = T\}$
var n: entero **fvar**
$n := 1$;
$r := 0$;
$\{I_1 \wedge I_2\}$
mientras $n \neq N$ **hacer**
$\{I_1 \wedge I_2 \wedge B\}$
'restablecer'
$\{(I_1 \wedge I_2)^n_{n+1}\}$
$n := n + 1$
$\{I_1 \wedge I_2\}$
fmientras
$\{Q : r = (\#i, j : 1 \leq i < j \leq N : t[i] < 0 \wedge t[j] > 0) \wedge t = T\}$

Para calcular las sentencias de 'restablecer' escribimos los predicados que determinan su funcionamiento,

$$I_1 : r = (\#i, j : 1 \leq i < j \leq n : t[i] < 0 \wedge t[j] > 0)$$
$$I_2 \wedge B : 1 \leq n < N$$

$$(I_1)^n_{n+1} : r = (\#i, j : 1 \leq i < j \leq n+1 : t[i] < 0 \wedge t[j] > 0)$$
$$(I_2)^n_{n+1} : 1 \leq n+1 \leq N$$

Observamos que $[[I_2 \wedge B \Rightarrow 1 \leq n+1 \leq N]]$ se cumple. El nuevo valor de r 'cambia' con respecto al anterior valor. Debemos estudiar la relación entre el predicado $(\#i, j : 1 \leq i < j \leq n+1 : t[i] < 0 \wedge t[j] > 0)$ y el anterior $(\#i, j : 1 \leq i < j \leq n : t[i] < 0 \wedge t[j] > 0)$.

$(\#i, j : 1 \leq i < j \leq n+1 : t[i] < 0 \wedge t[j] > 0)$
= (separar por el último término j = n + 1)
$= (\#i, j : 1 \leq i < j \leq n : t[i] < 0 \wedge t[j] > 0) + (\#i : 1 \leq i < n+1 : t[i] < 0 \wedge t[n+1] > 0)$
= (asumiendo $I_1 \wedge I_2 \wedge B$)
$= r + (\#i : 1 \leq i < n+1 : t[i] < 0 \wedge t[n+1] > 0)$
= (analizando los casos en el segundo término)

(caso 1) $= r + (\#i : 1 \leq i < n+1 : t[i] < 0)$ si $t[n+1] > 0$
(caso 2) $= r + 0$ si $t[n+1] \leq 0$

Formalmente, necesitamos restablecer con una sentencia alternativa con dos casos exclusivos,

$\{I_1 \wedge I_2 \wedge B\}$
si
[] $t[n+1] > 0 \rightarrow \{I_1 \wedge I_2 \wedge B \wedge t[n+1] > 0\}$ $r := r + (\#i : 1 \leq i < n+1 : t[i] < 0)$
[] $t[n+1] \leq 0 \rightarrow \{I_1 \wedge I_2 \wedge B \wedge t[n+1] \leq 0\}$ continuar
fsi;

$\{(I_1)^n_{n+1} \wedge (I_2)^n_{n+1}\}$

Claramente, no podemos programar la primera rama de la alternativa porque $(\#i : 1 \leq i < n+1 : t[i] < 0)$ debe ser calculado. Es en este punto dónde debemos decidir qué hacer a continuación. Comentamos las posibles soluciones:

- a) Realizar un programa para calcular $(\#i : 1 \leq i < n+1 : t[i] < 0)$ dejando su valor en una variable s, y cuya especificación sería,
  $\{1 \leq n < N\}$
  calcular
  $\{s = (\#i : 1 \leq i < n+1 : t[i] < 0)\}$
- b) Reforzar el invariante con la condición $s = (\#i : 1 \leq i < n+1 : t[i] < 0)$

En el primer caso tienes otro bucle con toda su derivación. Vamos a probar con la segunda posibilidad. Aquí de nuevo tenemos una elección que realizar y esta depende de la iniciación del programa. Recuerda que n comienza en 1, así que si n vale 1, s valdrá 1 o 0 dependiendo de si $t[1] < 0$ o no. Para simplificar la iniciación podemos probar como reforzamiento el predicado,

$$I_3 : s = (\#i : 1 \leq i < n : t[i] < 0)$$

cuya iniciación es más simple, puesto que cuando n vale 1, el rango se hace vacío y s vale 0. Además, lo que necesitamos para restablecer r en la primera rama de la alternativa anterior se obtiene del predicado $(I_3)^n_{n+1} : s = (\#i : 1 \leq i < n+1 : t[i] < 0)$. Efectivamente,

$(\#i, j : 1 \leq i < j \leq n+1 : t[i] < 0 \wedge t[j] > 0)$
$=$ (separar por el último término j = n + 1)
$=$ (asumiendo $I_1 \wedge I_2 \wedge B$)
$= r + (\#i : 1 \leq i < n+1 : t[i] < 0 \wedge t[n+1] > 0)$
$=$ (analizando los casos en el segundo término)

(caso 1) $= r + (\#i : 1 \leq i < n+1 : t[i] < 0)$ si $t[n+1] > 0$
(caso 2) $= r + 0$ si $t[n+1] \leq 0$

(asumendo ahora $(I_3)^n_{n+1}$)

(caso 1) $= r + s$ si $t[n+1] > 0$
(caso 2) $= r$ si $t[n+1] \leq 0$

Lo que esto implica es que antes de restablecer el valor de r tienes que calcular el nuevo valor de s en $(I_3)^n_{n+1}$ a partir de su valor en I_3. Veamos como,

$(\#i : 1 \leq i < n+1 : t[i] < 0)$
$=$ (separando por el último término)
$= (\#i : 1 \leq i < n : t[i] < 0) + \#(t[n] < 0)$
$=$ (analizando por casos y asumiendo $I_3 \wedge I_2 \wedge B$)

(caso 1) $= s + 1$ si $t[n] < 0$
(caso 2) $= s + 0$ si $t[n] \geq 0$

Por tanto el invariante que proponemos finalmente es,

$I_1 : r = (\#i, j : 1 \leq i < j \leq n : t[i] < 0 \wedge t[j] > 0)$
$I_2 : 1 \leq n \leq N$
$I_3 : s = (\#i : 1 \leq i < n : t[i] < 0)$
$I_1 \wedge I_2 \wedge I_3$ propuesta de invariante con reforzamiento

(5) Programa y terminación. El programa que resulta de toda la derivación anterior es el siguiente (están anotados los conjuntos de estados de la derivación)

constante
  N ($N \geq 1$)
fconstante
tipo
  tabla1N: **tabla** [1..N] **de** entero
ftipo
var
  t: tabla1N;
  r: entero
fvar
$\{P : t = T\}$
var n, s: entero **fvar**
$n := 1;$
$r := 0;$
$s := 0;$
$\{I_1 \wedge I_2 \wedge I_3\}$ // $cota : N - n$
mientras $n \neq N$ **hacer**
  $\{I_1 \wedge I_2 \wedge B \wedge I_3\}$
  si
    [] $t[n] < 0 \rightarrow s := s + 1$
    [] $t[n] \geq 0 \rightarrow$ continuar
  fsi;
  $\{I_1 \wedge I_2 \wedge B \wedge (I_3)^n_{n+1}\}$
  si
    [] $t[n+1] > 0 \rightarrow r := r + s$
    [] $t[n+1] \leq 0 \rightarrow$ continuar
  fsi;
  $\{(I_1)^n_{n+1} \wedge (I_2)^n_{n+1} \wedge (I_3)^n_{n+1}\}$
  $n := n + 1$
  $\{I_1 \wedge I_2 \wedge I_3\}$
fmientras
$\{Q : r = (\#i, j : 1 \leq i < j \leq N : t[i] < 0 \wedge t[j] > 0) \wedge t = T\}$

El programa anterior termia porque $f() = N - n$ es una función de cota. Lo importante es que la complejidad de este programa es $\mathcal{O}(N)$ y el reforzamiento ha jugado su papel en esta complejidad. □

7. Invariantes para funciones recursivas

Algunas funciones se pueden expresar de manera recursiva (recurrente). En esta sección discutimos unas técnicas muy básicas para diseñar programas que obtengan el resultado de

dichas funciones recursivas para unos datos de entrada dados. El primer ejemplo consiste en calcular el máximo común divisor de dos números positivos sin emplear la operación de división[5].

Ejemplo 8 *Máximo común divisor.*

El máximo común divisor de dos números positivos x e y, $\mathrm{mcd}(x, y)$, tiene las propiedades siguientes,

$$\begin{array}{ll} \mathrm{mcd}(x,y) = x & \text{si } x = y \\ \mathrm{mcd}(x,y) = \mathrm{mcd}(x-y,y) & \text{si } x > y \\ \mathrm{mcd}(x,y) = \mathrm{mcd}(x,y-x) & \text{si } y > x \end{array}$$

Aplicando está fórmula podemos obtener el mcd de dos números positivos sin hacer ninguna división:

$$\begin{aligned} \mathrm{mcd}(60,144) &= \mathrm{mcd}(60,144-60) = \mathrm{mcd}(60,84-60) = \mathrm{mcd}(60-24,24) \\ &= \mathrm{mcd}(36-24,24) = \mathrm{mcd}(12,24-12) = \mathrm{mcd}(12,12) = 12 \end{aligned}$$

Sobre el mismo ejemplo podemos deducir el invariante del proceso iterativo llevado a cabo. Si en todo momento x e y toman los valores de los argumentos de la función mcd() en cada una de las igualdades anteriores, podemos decir que, 'lo que queremos calcular al final $\mathrm{mcd}(60, 144)$ es igual a lo que me falta de calcular en las variables x e y, $\mathrm{mcd}(x, y)$'. La propia definición de la función mcd() nos dá todo lo que necesitamos hacer. La condición de salida es $x = y$, la condición de continuación es $x \neq y$. La forma de avanzar será $x := x$ - y si $x > y$, o bien $y := y$ - x si $y > x$. Así, que dada la especificación siguiente,

var
x, y: entero
fvar
$\{P : x = A \wedge y = B \wedge x > 0 \wedge y > 0\}$
'máximo común divisior'
$\{Q : x = \mathrm{mcd}(A, B)\}$

Utilizando, las propiedades de la función mcd() dadas, la propuesta de invariante es,

$$I : \mathrm{mcd}(A,B) = \mathrm{mcd}(x,y) \wedge x > 0 \wedge y > 0$$

(1) Condición de salida. $[[I \wedge \neg B \Rightarrow Q]]$. Si $\neg B$ es $x = y$, entonces $\mathrm{mcd}(A, B) = \mathrm{mcd}(x, y) \wedge x = y$ es equivalente a $\mathrm{mcd}(A, B) = \mathrm{mcd}(x, x) \wedge x = y$, con lo que $\mathrm{mcd}(A, B) = x$ se cumple. La condición de continuación B es $x \neq y$.

(2) Instrucciones de inicio. Las variables x e y ya toman un valor al principio, y se cumple directamente que $[[P \Rightarrow I]]$. Así, que no se necesitan ninguna instrucción de inicio.

(3) Avanzar a la terminación. La condición de continuación es $x \neq y$ por lo que o bien hacemos que x vaya hacia y o al revés. La propia definición de la función nos indica que si $x > y$ hay que hacer $x := x$ - y, dejando a y sin modificar. Lo mismos sucede en el caso $y > x$ de manera simétrica.

[5]El algoritmo de Euclides emplea la operación de división y es el algoritmo más eficiente para realizar el cálculo del máximo común divisor.

(4) Restablecer el invariante. Observa lo siguiente en el caso $x > y$,

$$
\begin{array}{l}
(I)^{x}_{x-y} : \mathrm{mcd}(A,B) = \mathrm{mcd}(x-y,y) \wedge x - y > 0 \wedge y > 0 \\
\equiv \\
\mathrm{mcd}(A,B) = \mathrm{mcd}(x-y,y) \wedge x > y \wedge y > 0 \\
\equiv \quad \text{(por el caso de la definición de mcd())} \\
\mathrm{mcd}(A,B) = \mathrm{mcd}(x,y) \wedge x > y \wedge y > 0 \\
\equiv \\
I \wedge x > y
\end{array}
$$

De forma simétrica se cumple $[[I \wedge y > x \equiv (I)^{y}_{y-x}]]$. Como para $x \neq y$, $x > y$ e $y > x$ son exclusivos, la sentencia alternativa siguiente constituye el restablecer del bucle:

$\{I \wedge x \neq y\}$
si
  [] $x > y \rightarrow \{I \wedge x > y\}$ x := x - y
  [] $y > x \rightarrow \{I \wedge y > x\}$ y := y - x
fsi;
$\{I\}$

(5) Programa y terminación. El programa obtenido de la derivación anterior es,

var
  x, y: entero
fvar
$\{P : x = A \wedge y = B \wedge x > 0 \wedge y > 0\}$
$\{I : \mathrm{mcd}(A,B) = \mathrm{mcd}(x,y) \wedge x > 0 \wedge y > 0\}$ *//cota* : $x + y$
mientras $x \neq y$ **hacer**
  si
    [] $x > y \rightarrow x$:= x - y
    [] $y > x \rightarrow y$:= y - x
  fsi;
fmientras
$\{Q : x = \mathrm{mcd}(A,B)\}$

Los estudiantes pueden comprobar que una función de cota que garantiza la terminación es $f() = x + y$. □

Ejemplo 9 *Potencia de dos números enteros.*

Dados dos números enteros x e y con $y \geq 0$, la potencia x^y, cumple lo siguiente,

$$
\begin{array}{ll}
x^y = 1 & \text{si } y = 0 \\
x^y = x\, x^{y-1} & \text{si } y > 0
\end{array}
$$

Aplicado estas fórmulas podemos obtener 3^4 de la siguiente manera,

$$3^4 = 3 \times 3^3 = 3 \times 3 \times 3^2 = 3 \times 3 \times 3 \times 3^1 = 3 \times 3 \times 3 \times 3 \times 3^0 = 3 \times 3 \times 3 \times 3 \times 1$$

Consideremos la siguiente especificación,

var
  x, y, p: entero

fvar
$\{P : x = A \wedge y = B \wedge y \geq 0\}$
'potencia de dos enteros'
$\{Q : p = A^B\}$

En el ejemplo anterior, si tomamos cualquier igualdad, como por ejemplo, $3^4 = 3 \times 3 \times 3^2$ vemos que lo que queremos calcular 3^4 es igual a lo que nos resta de calcular 3^2 multiplicado por lo que vamos acumulando del producto 3×3. Las variables x e y en todo momento indican la base y el exponente respectivamente, y p puede ir recogiendo las sucesivas aproximaciones a la potencia. Entonces el invariante para este programa puede ser tan simple como,

$$I : A^B = p\, x^y \wedge y \geq 0$$

Los estudiantes pueden derivar la siguiente solución,

var
x, y, p: entero
fvar
$\{P : x = A \wedge y = B \wedge y \geq 0\}$
$p := 1;$
$\{I : A^B = p\, x^y \wedge y \geq 0\}$ $//cota : y$
mientras $y \neq 0$ **hacer**
$p := p$ * $x;$
$y :=$ y - 1
fmientras
$\{Q : p = A^B\}$

Ejemplo 10 *Potencia eficiente de dos números enteros.*

El número de vueltas del bucle en la solución anterior es B, ya que y comienza en B y termina siendo 0. Así, la complejidad es $\mathcal{O}(B)$ (lineal). La instrucción que avanza a la terminación es $y :=$ y - 1. Ahora bien, ¿qué es más rápido que restar?...uhmm...dividir. ¿Por qué no avanzar dividiendo por un número?. Si esto se puede hacer así, entonces se debe reflejar en las propias propiedades de la función potencia. El número más sencillo para dividir es dos, con la ventaja de que distingue a los números enteros en pares o impares. Podemos comprobar que la potencia cumple las siguientes propiedades,

$$\begin{array}{ll} x^y = 1 & \text{si } y = 0 \\ x^y = (x^2)^{(y \text{ div } 2)} & \text{si } y > 0 \wedge y \bmod 2 = 0 \text{ (y es par)} \\ x^y = x\,(x^2)^{(y \text{ div } 2)} & \text{si } y > 0 \wedge y \bmod 2 = 1 \text{ (y es impar)} \end{array}$$

Aplicado esas fórmulas podemos obtener 3^{28} de la siguiente manera,

$$3^{28} = (3^2)^{(14)} = (3^4)^{(7)} = 3^4 \times (3^8)^{(3)} = 3^4 \times 3^8 \times (3^{16})^{(1)} = 3^4 \times 3^8 \times 3^{16} \times 1$$

De nuevo podemos utilizar el invariante anterior

$$I : A^B = p\, x^y \wedge y \geq 0$$

Pero en este caso, en cada paso del bucle la base x se incrementa en x^2 mientras que el exponente en y pasa a ser y div 2. Si hemos realizado la derivación del ejercicio anterior la condición de continuación del bucle es $y \neq 0$, y una función de cota para el bucle es $f() = y$, ya que $I \wedge B$

implica $y > 0$. Para avanzar en el caso que nos ocupa, debemos divir por 2, haciendo $y := y$ div 2. Para restablecer el invariante, la propia definición recurrente dada de la potencia sugiere una alternativa con dos casos. Observamos lo siguiente,

$$
\begin{array}{l}
I \wedge B \wedge y \bmod 2 = 0 \text{ (y es par)} \\
\equiv \\
A^B = p\,x^y \wedge y > 0 \wedge y \bmod 2 = 0 \\
\equiv \text{(por el segundo caso de la definición de la función potencia)} \\
A^B = p\,(x^2)^{(y \text{ div } 2)} \wedge y > 0 \wedge y \bmod 2 = 0 \\
\Rightarrow \\
A^B = p\,(x^2)^{(y \text{ div } 2)} \wedge y \text{ div } 2 \geq 0 \\
\equiv \\
((I)^y_{y\,div\,2})^x_{x*x}
\end{array}
$$

para el caso impar,

$$
\begin{array}{l}
I \wedge B \wedge y \bmod 2 \neq 0 \text{ (y es impar)} \\
\equiv \\
A^B = p\,x^y \wedge y > 0 \wedge y \bmod 2 = 1 \\
\equiv \text{(por el tercer caso de la definición de la función potencia)} \\
A^B = p\,x\,(x^2)^{(y \text{ div } 2)} \wedge y > 0 \wedge y \bmod 2 = 1 \\
\Rightarrow \\
A^B = p\,x\,(x^2)^{(y \text{ div } 2)} \wedge y \text{ div } 2 \geq 0 \\
\equiv \\
(((I)^y_{y\,div\,2})^x_{x*x})^p_{p*x}
\end{array}
$$

Por lo que el algoritmo nos queda como,

var
  x, y, p: entero
fvar
$\{P : x = A \wedge y = B \wedge y \geq 0\}$
$p := 1;$
$\{I : A^B = p\,x^y \wedge y \geq 0\}$ $//cota : y$
mientras $y \neq 0$ **hacer**
  si
    [] $y \bmod 2 = 0 \rightarrow$ continuar
    [] $y \bmod 2 \neq 0 \rightarrow p := p * x$
  fsi;
  $x := x * x;$
  $y :=$ y div 2
fmientras
$\{Q : p = A^B\}$

La complejidad en tiempo de este algoritmo es $\mathcal{O}(\log_2(B))$, es decir, de orden logarítmico. Calcular la potencia 3^{1024} requiere en el primer algoritmo dar 1024 vueltas en el bucle. Mientras que con el algortimo eficiente sólo se dan 10 vueltas!!.

8. Ejercicios

Los ejercicios para la primera evaluación son los que se encuentran propuestos en las primeras cinco Lecturas y los que se han enunciado y resuelto en los guiones de las prácticas de las sesiones

correspondientes. Cada ejercicio se debe resolver siguiendo el método de derivación de bucles mediante invariantes. Los ejercicios deben acompañarse de los cálculos lógicos realizados en la derivación, incluida la demostración para la función de cota. Es aconsejable realizar algunas ejecuciones de los algoritmos obtenidos para asegurase que la derivación ha sido bien realizada. También, se debe calcular la complejidad del algoritmo obtenido. Los criterios de corrección, para los exámenes de los ejercicios de la parte de teória, se encuentran en el documento correspondiente de miaulario, en el sitio de la asignatura, en la carpeta 'criterios de corrección'. Finalmente, los ejercicios de los exámenes pueden coincidir o no con los enunciados dados en las lecturas, en todo caso, siempre serán del mismo grado de dificultad que los propuestos en ellas.

Nota: En algunos de los ejercicios se usan cuantificadores pero las variables ligadas no se nombran con letras griegas. A estas alturas del curso los estudiantes deben saber diferenciar las variables ligadas de aquellas que están libres. Tenga en cuenta, que en las soluciones, las sentencias no deben contener nombres de variables ligadas. En caso contrario, no se podrían realizar las demostraciones de verificación!.

1. **Factorial**

   El factorial de un número n entero positivo se define recursivamente

   $$n! = 1 \text{ si } n = 0$$
   $$n! = 1 \text{ si } n = 1$$
   $$n! = n \times (n-1)! \text{ si } n \geq 1$$

   Diseñe un algoritmo con un solo bucle que resuelva el siguiente problema

   n, fact: *entero*
   $\{P : n = N \wedge N > 0\}$
   factorial
   $\{Q : fact = N!\}$

   Siendo el invariante del bucle el predicado siguiente

   $$I : fact = \frac{(i-1)!n!}{j!} \wedge 0 < i \leq j+1 \wedge n = N$$

   El número de iteraciones del bucle no debe superar n div $2+1$. Proponga una función de cota.

2. **Suma de coeficientes binomiales**

   Sea la función definida recurrentemente para un número entero n positivo

   $$C(n,0) = 1$$
   $$C(n,k+1) = C(n,k) \times \tfrac{n-k}{k+1} \ , \qquad 0 \leq k < n$$

   Diseñe un algoritmo con un solo bucle que resuelva el siguiente problema

   n, r: *entero*;
   $\{P : n = N \wedge N \geq 1\}$
   suma_de_coeficientes_binomiales
   $\{Q : n = N \wedge r = \sum_{i=0}^{n} C(n,i)\}$

   Siendo el invariante del bucle el predicado siguiente

   $$I : r = \textstyle\sum i : 0 \leq i \leq j : C(n,i) \wedge n = N \wedge 0 \leq j \leq n \wedge c = C(n,j)$$

Proponga una función de cota.

3. **Seno(x)**

   Dada la especificación

   x, s: *real*
   n: *entero*
   $\{P : x = X \wedge -\pi \leq X < \pi \wedge\ n = N \wedge\ N \geq 0\}$
   seno_de_x
   $\{Q : s = \sum_{0 \leq i \leq n} (-1)^i \cdot \frac{x^{2 \cdot i+1}}{(2 \cdot i+1)!} \wedge\ x = X \wedge\ n = N\}$

   Derive un algoritmo basado en un único bucle que resuelva el problema. Se debe utilizar como invariante para el bucle el predicado siguiente

   $$I : s = \sum_{0 \leq i \leq k} (-1)^i \cdot \frac{x^{2 \cdot i+1}}{(2 \cdot i+1)!} \ \wedge\ x = X \wedge\ n = N \wedge\ 0 \leq k \leq N \wedge$$
   $$\wedge\ p = x^{2 \cdot k+1} \wedge\ q = (2 \cdot k+1)! \wedge\ r = (-1)^k$$

   Proponga una función de cota.

4. **Factorial inferior**

   Diseñe un algoritmo con un solo bucle que resuelva el siguiente problema

   x, n: *entero*
   $\{P : x = X \wedge\ X \geq 1\}$
   factorial_inferior
   $\{Q : x = X \wedge\ n! \leq x < (n+1)!\}$

   Siendo el invariante del bucle el predicado siguiente

   $$I : x = X \wedge\ n! \leq x \wedge\ f = (n+1)!$$

   Proponga una función de cota.

5. **Fibonacci inferior**

   La serie de Fibonacci se define

   $$fib(0) = 1$$
   $$fib(1) = 1$$
   $$fib(i) = fib(i-1) + fib(i-2)\ , \qquad i \geq 2$$

   Diseñe un algoritmo con un solo bucle que resuelva el siguiente problema

   n, i: *entero*;
   $\{P : n = N \wedge\ N > 0\}$
   fibonacci_inferior
   $\{Q : n = N \wedge\ fib(i) \leq N < fib(i+1)\}$

   Siendo el invariante del bucle el predicado siguiente

   $$I : n = N \wedge\ fib(i) \leq n \wedge\ f = fib(i) \wedge\ g = fib(i+1)$$

   Proponga una función de cota.

6. **Raíz aproximada**

   Dada la especificación

   n, r: *entero*;
   $\{P : n = N \ \wedge \ N \geq 1\}$
   raíz_aproximada
   $\{Q : r^3 \leq N < (r+1)^3\}$

   Derive un algoritmo basado en un único bucle que resuelva el problema. Se debe utilizar como invariante para el bucle el predicado siguiente

   $$I : r^3 \leq N \wedge \ n = N \wedge \ a = (r+1)^3 \wedge \ b = (r+1)^2$$

   Para este ejercicio, no está permitido el producto de dos variables, pero sí está permitido el producto de una constante por una variable. Por ejemplo no está permitido $x * x$ ni $x * x * x$, pero sí está permitido $2 * x$ ó $3 * x$. Proponga una función de cota.

7. **Mayor índice para una función**

   Sea la siguiente función $A()$, definida como

   $$\begin{aligned} A(0) &= 1 \\ A(i) &= 2 \cdot A(i \text{ div } 2) \ , \quad \text{para } i \text{ par y } i > 0 \\ A(i) &= (i \text{ div } 2) + A(i-1) \ , \quad \text{para } i \text{ impar y } i \geq 0 \end{aligned}$$

   Diseñe un algoritmo con un solo bucle que resuelva el siguiente problema

   n, i: *entero*;
   $\{P : n = N \wedge 1 \leq N \leq 99\}$
   rango
   $\{Q : n = N \wedge A(i) \leq n < A(i+1)\}$

   Siendo el invariante del bucle el predicado siguiente

   $$I : n = N \wedge \ A(i) \leq n \wedge \ f = A(i+1) \wedge \ (\forall j : 0 \leq j \leq i : t[j] = A(j))$$

   Teniendo en cuenta que $i \leq A(i)$, se ha dispuesto una tabla auxiliar

   t: **tabla**[0..99] **de** *entero*

   para guardar los valores de la función $A()$ que se han calculado en anteriores iteraciones.

   Proponga una función de cota.

8. **Logaritmo en base dos**

   Cuenta la leyenda que, hace muchos años, el maharajá Sirham de la India quiso recompensar a su visir Sissa Ben Dahir el que éste hubiera inventado para el rey el juego del ajedrez, y dejó que el visir pidiera la recompensa. Éste solicitó que se le entregara la cantidad de granos de trigo que resultara de sumar 2^0 granos colocados en la primera casilla, 2^1 colocados en la segunda, 2^2 colocados en la tercera y así sucesivamente hasta 2^{63} correspondientes a la casilla número 64, última del tablero. El rey aceptó la petición —¡no pides mucho, mi fiel servidor, tu deseo será cumplido!. Parece que esa noche el rey puso a trabajar a los contables y a la mañana siguiente ordenó la detección y ejecución de su visir. ¿Qué conoció el rey para tomar tal decisión?. Hoy sabemos que

   $$2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{63} = 2^{64} - 1$$

y que ese número, para darnos idea de su tamaño, escrito con todos sus dígitos es el

$$18_3{,}446{,}744_2{,}073{,}709_1{,}551{,}615$$

número pues de granos de trigo de la deuda.

Teniendo en cuenta que al año se producen unos 10^{16} granos de trigo, serían necesarios ¡1845 años de producción! para saldar la deuda[6].

La historia podría haber acabado mejor (¡sobre todo para el visir!) si ambos hubieran reconocido sus límites. Así, si el rey hubiera estado dispuesto a entregar M granos de trigo, su visir podría haber pedido que rellenaran hasta la casilla r en lugar de todas las del tablero, procurando no superar la cantidad prevista por el rey. Como el cálculo es pesado, se trata de desarrollar un algoritmo que ayude al visir en esta labor. El algoritmo debe cumplir la especificación

m, r: *entero*
$\{P : m = M \ \wedge \ M \geq 1\}$
hasta_donde_puedo_llegar
$\{Q : \sum_{0 \leq k \leq r} 2^k \leq m < \sum_{0 \leq k \leq r+1} 2^k \wedge \ m = M\}$

Derive un algoritmo basado en un único bucle que resuelva el problema. Se debe utilizar como invariante para el bucle el predicado siguiente

$$I : \sum_{0 \leq k \leq r} 2^k \ \leq \ m \ \wedge \ m = M \ \wedge \ q = \sum_{0 \leq k \leq r} 2^k \ \wedge \ p = 2^{r+1}$$

Proponga una función de cota.

9. **Suma productos de potencias**

Dada la especificación

x, y, z, n, s: *entero*;
$\{Pre : x = X \wedge y = Y \wedge z = Z \wedge X > 1 \wedge Z \geq 1\}$
suma_de_productos
$\{Post : x = X \wedge y = Y \wedge z = Z \wedge s = \sum_{0 \leq k \leq n} x^k \cdot y^{n-k} \wedge x^n \leq z < x^{n+1}\}$

Construye un algoritmo que resuelva el problema. Con este fin debes considerar el problema como la unión secuencial de los dos subproblemas siguientes

$\{P_1 : x = X \wedge y = Y \wedge z = Z \wedge X > 1 \wedge Z \geq 1\}$
$\{Q_1 : P_1 \wedge x^n \leq z < x^{n+1} \wedge p = x^n\}$

$\{Q_1\}$
$\{Q_2 : x = X \wedge y = Y \wedge z = Z \wedge s = \sum_{0 \leq k \leq n} x^k \cdot y^{n-k} \wedge x^n \leq z < x^{n+1}\}$

Es necesario que propongas un invariante Inv_1 para resolver el primer subproblema, mientras que deberás utilizar como invariante para resolver el segundo subproblema el predicado siguiente

[6]Pasaje extraído del libro *Año Mundial de las Matemáticas*, G. Ochoa y J.R. Pascual (Eds.), Universidad Pública de Navarra, 2002, págs. 32-33.

$$Inv_2 : x = X \wedge y = Y \wedge z = Z \wedge s = \sum_{i \le k \le n} x^k \cdot y^{n-k} \wedge$$
$$\wedge\, x^n \le z < x^{n+1} \wedge 0 \le i \le n \wedge p = x^i \wedge q = y^{n-i}$$

Proponga funciones de cota para los dos bucles.

10. **Suma de factoriales**

Dada la especificación

x, n, s: *entero*;
$\{P : x = X \wedge X \ge 1\}$
suma_de_factoriales
$\{Q : s = \sum_{1 \le i \le n} i! \wedge n! \le x < (n+1)! \wedge x = X\}$

Construye un algoritmo que resuelva el problema con un único bucle. Define el invariante para la solución y proponga una función de cota.

11. **Coeficientes binomiales**

El número de subconjuntos de m elementos que se pueden hacer en un conjunto de cardinal n, $\binom{n}{m}$, verifica la igualdad

$$\binom{n}{m} = \frac{n!}{m! \cdot (n-m)!}$$

Dada la especificación

n, m, q: *entero*;
$\{P : n = N \wedge\ m = M \wedge\ N \ge M \ge 0\}$
coeficientes_binomiales
$\{Q : q = \binom{n}{m} \wedge\ n = N \wedge\ m = M\}$

Derive un algoritmo basado en un único bucle que resuelva el problema. Se debe utilizar como invariante para el bucle el predicado siguiente

$$Inv : q = \frac{n!}{i! \cdot (n-i)!} \wedge\ 0 \le i \le m \wedge\ n = N \wedge\ m = M$$

Proponga una función de cota.

a) Realice el mismo ejercicio pero utilizando la sustitución de n por una nueva variable. Escriba el invariante correspondiente y realice la derivación del algoritmo.
b) A partir de las dos soluciones anteriores y dados n y m, encuentre la solución con el coste computacional más bajo.

12. **Cálculo de aproximaciones de π**

La razón entre la longitud de una circunferencia y su diámetro es una constante real a la que se ha convenido en nombrar como π. El número π es transcendente sobre el cuerpo de los racionales $\mathcal{Q}$ por lo que son necesarios infinitos valores enteros para poder expresarlo de forma exacta. Una de las expresiones que se conocen para el valor π aparece en la igualdad

$$\pi = 4 \cdot \sum_{i \in \mathbb{N}} \frac{(-1)^i}{2 \cdot i + 1}$$

Derive un algoritmo que satisfaga la especificación

n, k: *entero*;
p: *real*;
$\{Pre \equiv n = N \ \wedge \ N \geq 0\}$
aproximación_de_π
$\{Post \equiv n = N \ \wedge \ p = 4 \cdot \sum_{0 \leq i \leq k} \frac{(-1)^i}{2 \cdot i + 1} \ \wedge$
$\wedge \ \frac{1}{2 \cdot k + 3} < \frac{1}{10^n} \leq \frac{1}{2 \cdot k + 1}\}$

Puede utilizar la función potencia para calcular 10^n.

13. **Otro máximo común divisor**

Derive un algoritmo iterativo para el cálculo del máximo común divisor basado en las siguientes relaciones: Sean x e y dos enteros tales que $x, y > 0$

a) m.c.d.(x, y) = m.c.d.(y, x);
b) m.c.d.$(x, y) = x$, si $x = y$;
c) m.c.d.$(x, y) = 2 \times$ m.c.d.$(x/2, y/2)$, si $x > y$, x e y son pares;
d) m.c.d.(x, y) = m.c.d.$(x/2, y)$, si $x > y$, x es par, e y es impar;
e) m.c.d.(x, y) = m.c.d.$(x, y/2)$, si $x > y$, x es impar, e y es par;
f) m.c.d.(x, y) = m.c.d.$((x + y)/2, (x - y)/2)$, si $x > y$, x e y son impares.

La especificación del problema es

x, y: *entero*;
$\{P : x = X \ \wedge \ y = Y \ \wedge \ X \geq Y > 0\}$
máximo_común_divisor
$\{Q : x = \text{m.c.d.}(X, Y)\}$

Utilizando como invariante para el bucle del algoritmo el siguiente predicado

$$\text{Inv} \equiv x \geq y > 0 \ \wedge \ \text{m.c.d.}(X, Y) = n \times \text{m.c.d.}(x, y)$$

Proponga una función de cota. Presente los cáculos de la derivación.

14. **Regla de integración de Simpson**

La regla de Simpson es un método de integración numérica. Usando la regla de Simpson, el valor de la integral entre a y b de una función real de una variable f es aproximado mediante el valor

$$\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 4y_{n-3} + 2y_{n-2} + 4y_{n-1} + y_n)$$

donde n es un entero par mayor que cero, $h = (b - a)/n$ e $y_k = f(a + k \cdot h)$, para cada entero k entre 0 y n.

Supóngase que la constante N se ha definido a un valor entero par mayor que cero. Se pide diseñar un algoritmo que satisfaga la especificación

constante

N ($N \geq 1$)
fconstante
tipo
tabla0N: **tabla** [0..N] **de** real
ftipo
var
t: tabla0N;
a, b, r: real
fvar
$\{P : t = T \ \wedge \ a = A \ \wedge \ b = B \ \wedge \ N$ es par $\ \wedge \ N > 0 \ \wedge \ (\forall k : 0 \leq k \leq N : t[k] = y_k)\}$
'Simpson'
$\{Q : P \wedge \ r = \dfrac{b-a}{3N}(y_0 + 4y_1 + 2y_2 + \cdots + 4y_{N-1} + y_N)\}$

y utilice una única composición iterativa cuya acción esté descrita por el siguiente predicado invariante:

$$\text{Inv} : r = y_0 + \sum_{1 \leq k \leq j} (4y_{2k-1} + 2y_{2k}) \ \wedge \ 0 \leq j \leq N \ \textbf{div} \ 2$$

15. **Partición por un valor**

Construya un algoritmo que, dada una tabla de enteros de N posiciones y un valor, indique la partición de la tabla cuyos elementos suman un valor inferior o igual al dado, según la especificación siguiente

t: **tabla** [1..N] **de** *entero*;
x, p: *entero*;
$\{Pre : t = T \ \wedge \ x = X \ \wedge \ (\forall k : 1..N : t[k] \geq 0) \ \wedge \ 0 \leq x < (\sum k : 1..N : t[k])\}$
partición_por_x
$\{Post : t = T \ \wedge \ x = X \ \wedge \ (\sum k : 1..p : t[k]) \leq x < (\sum k : 1..p+1 : t[k]) \wedge$
$\wedge \ 0 \leq p < N\}$

En este ejercicio hay que definir un invariante para resolver el problema mediante el método de 'elección de una conjunción'. Derive la solución según el invariante propuesto y defina una función de cota.

16. **Posición del mínimo de los valores almacenados en una tabla**

Diseñe un algoritmo que resuelva el siguiente problema

t: **tabla** [1..N] **de** *entero*;
i: 1..N;
$\{Pre : t = T \wedge \ (\forall k, l : 1 \leq k, l \leq N \wedge \ k \neq l : t[k] \neq t[l])\}$
mínimo_de_tabla
$\{Post : t = T \wedge \ t[i] = \min\{t[k] : 1 \leq k \leq N\}\}$

El algoritmo ha de construirse empleando una única composición iterativa cuyo invariante es

$$\text{Inv} : t = T \wedge \ 1 \leq i \leq j \leq N \wedge \ \min\{t[k] : i \leq k \leq j\} = \min\{t[k] : 1 \leq k \leq N\}$$

Derive la solución y proponga una función de cota.

17. **Cuenta de parejas con el mismo signo**

Diseñe un algoritmo basado en un único bucle que resuelva el siguiente problema

t: **tabla** [1..N] **de** *entero*;
r: *entero*;
$\{Pre : t = T \wedge (\forall k : 1..N : t[k] \neq 0)\}$
    cuenta_parejas
$\{Post : t = T \wedge \ r = \#(i,j) : 1 \leq i < j \leq N : t[i] \cdot t[j] > 0\}$

El algoritmo debe constar de un sólo bucle. El método adecuado para resolver el problema es construir la solución mediante sustitución de constantes por variables y reforzar el invariante después de realizar la primera derivación de la solución.

18. **Máxima diferencia**

Diseñe un algoritmo basado en un único bucle que resuelva el siguiente problema

t: **tabla** [1..N] **de** *entero*;
r: *entero*;
$\{Pre : t = T\}$
    máxima_diferencia
$\{Post : t = T \wedge \ r = \max\{t[i] - t[j] : 1 \leq i < j \leq N\}\}$

La solución de este ejercicio basado en un único bucle tiene un poco más de dificultad, pero resulta interesante comprobar la ventaja de disponer de un método de derivación para conseguir algoritmos, en principio, nada evidentes.

Referencias

Me falta completar