Programación

Asignatura 240205 Programación. Curso 2020–2021

Lectura[4]: **Especificación y verificación de programas**

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

Resumen

Los programas se diseñan para cumplir con un propósito. El objetivo de un programa se especifica mediante las condiciones sobre los estados al comienzo de la ejecución en relación con las condiciones sobre los estados al final de la ejecución. En este curso utilizamos predicados para definir tales condiciones y la especificación se concreta dando un predicado inicial, Precondición, y un predicado final, Postcondición, sobre las variables que se van a utilizar en el programa. La corrección de un programa se establece cuando comprobamos que para todo estado que cumple la precondición, el programa termina; y termina en algún estado que cumple la postcondición. Se puede establecer un método para verificar si un programa cumple con su especificación a través de lo que denominamos 'reglas de inferencia'. Dichas reglas, establecidas para cada forma de composición de sentencias en un programa, nos permitirán verficar la validez de un programa. Se puede ir un paso más allá: conociendo las reglas de inferencia, éstas se pueden emplear para 'calcular' las sentencias que harán correcto un determinado programa. Derivar programas correctos y eficientes será el objetivo principal del curso de Programación.

Índice

1. Especificaciones

Un programa tiene un propósito y sus sentencias se construyen para obtener tal fin. La **especificación de un programa** se determina mediante una declaración de variables, una **precondición** y una **postcondición**. Por ejemplo, la siguiente especificación se presenta para un programa que debe calcular el máximo común divisor de dos número positivos X e Y:

1: **var**
2: x, y: entero ▷ declaración de variables
3: **fvar**
4: $\{P : x = X \wedge y = Y \wedge X > 0 \wedge Y > 0\}$ ▷ precondición
5: 'máximo común divisor' ▷ nombre del programa
6: $\{Q : x = mcd(X, Y)\}$ ▷ postcondición

Las variables x e y y su tipo (entero) determinan los posibles estados que podemos formar con dichas variable. Un estado concreto está definido por los valores que toman las variables en dicho estado. En el ejemplo, son los puntos de $\mathbb{Z} \times \mathbb{Z}$ dónde la primera coordenada corresponde a x y la segunda a y. Tanto en la precondición, $x = X \wedge y = Y \wedge X > 0 \wedge Y > 0$, como en la postcondición, $x = mcd(X, Y)$, X e Y representan valores de las variables x e y. Como estos valores son arbitrarios y no se conocen hasta que un programa, ya diseñado, los solicita al comienzo de su ejecución, se denominan **variables de la especificación**[1]. Sólo foman parte de las especificaciones y **no** se pueden emplear en las sentencias de los programas.

Considera que has diseñado un programa S para la especificación anterior. ¿Cómo sabes que es correcto?. El programa S **satisface** la especificación dada si para todos los enteros X e Y, la ejecución de S comenzando en un estado que cumple $x = X \wedge y = Y \wedge X > 0 \wedge Y > 0$ termina en un estado que cumple $x = mcd(X, Y)$.

Considera que S satisface su especificación $\{P\}$-$\{Q\}$. Si comenzamos en el estado $\sigma_i = \langle 12, 144\rangle$, S termina en un estado $\sigma_f = \langle 12, -\rangle$ ya que el estado incial σ_i cumple P y σ_f cumple Q, ya que $12 = mcd(12, 144)$. Oberva que la interpretación que hemos dado es una interpretación **operacional** (cómo funciona), y observa que está condicionada 'si...'. Si comienzas en un estado $\sigma_i = \langle -12, 144\rangle$ no cumple P (los números tienen que ser positivos), y por tanto, no se puede concluir nada sobre el comportamiento de S (no lo has diseñado para ese propósito).

Si tengo la especificación anterior $\{P\}$-$\{Q\}$ y he diseñado un nuevo programa S'. ¿Cómo puedo convencerme de que S' es correcto?. Si comienzas en un estado $\sigma_i = \langle 12, 144\rangle$ y obtienes $\sigma_f = \langle 12, -\rangle$. Para ese estado es correcto. Pero, si tu programa comienza en un estado $\sigma_i = \langle 7, 3\rangle$ y obtienes $\sigma_f = \langle 3, -\rangle$ inmediatamente te das cuenta de que no funciona bien ya que 7 y 3 son primos. Surge una pregunta obvia, ¿cuántos casos tengo que probar para comprobar que mi programa es correcto?. En algunas ocasiones no es posible probar todos los casos, y en esas circunstancias, 'la ausencia de errores en las pruebas realizadas, no impide que haya otra prueba en la que el programa falle'.

A lo largo del curso de Programación daremos reglas demostrativas (**reglas de inferencia**) que nos permitan asegurar que los programas satisfacen su especificación. Conociendo las **reglas de inferencia** también las usaremos para **derivar** (diseñar) programas correctos.

Una especificación también puede ser vista como un 'problema' que queremos resolver mediante un programa. De hecho la especificación indica cuáles son los estados iniciales y cuáles son los estados finales que obtendríamos al ejecutar un programa que cumple la especificación. Por ese motivo, las especifaciones de programas también se denominan **especificaciones de problemas algorítmicos**.

[1] Para distinguir las 'variables de la especificación' de las 'variables del programa', escribiremos siempre las primeras con letras mayúsculas. Las variables del programa se nombran mediante palabras escritas en letras minúsculas.

Ejemplo 1 División entera.

Dados dos números A y B siendo B el divisor, positivo distinto de cero, podemos encontrar otros dos números q y r tales que $A = B \cdot q + r$ con $0 \leq r \leq B$. Necesitamos al menos cuatro variables, x, y, q, r. Las dos primeras recogerán el dividendo y el divisor y en q y r dejamos el cociente y el resto.

1: **var**
2: x, y: entero
3: q, r: entero
4: **fvar**
5: $\{x = A \wedge y = B \wedge y > 0\}$ ▷ precondición
6: 'división entera' ▷ nombre del programa
7: $\{x = q \cdot y + r \wedge 0 \leq r < y \wedge x = A \wedge y = B\}$ ▷ postcondición

□

Ejemplo 2 Decidir si un número mayor que uno es primo o no.

Un número $N > 1$ es primo si no encontramos un divisior que sea diferente a 1 y N. Devolvemos la decisión de si es primo o no en una variable boleana.

1: **var**
2: n: entero
3: b: boleano
4: **fvar**
5: $\{n = N \wedge N > 1\}$ ▷ precondición
6: 'es primo?' ▷ nombre del programa
7: $\{b \equiv (\forall \gamma : 1 < \gamma < n : n \, mod \, \gamma \neq 0) \wedge n = N\}$ ▷ postcondición

□

Ejemplo 3 Posición del máximo en una tabla.

Disponemos de una tabla de N elementos. En cada elemento encontramos un número entero. Las posiciones de los elementos van desde 0 hasta $N-1$. N se determina como una constante que es mayor que uno (la tabla no está vacía). Queremos obtener la posición en la que se encuentra el valor máximo entre los valores de los elementos de la tabla.

1: **constante**
2: $N > 1$
3: **fconstante**
4: **var**
5: t: tabla$[0..N-1]$ de entero
6: $pmax$: entero
7: **fvar**
8: $\{t = T\}$
9: 'posición del máximo'
10: $\{t = T \wedge (\forall \gamma : 0 \leq \gamma < N : t[pmax] \geq t[\gamma]) \wedge 0 \leq pmax < N\}$

En la precondición, $t = T$ indica que la variable t contiene una tabla dada T, por ejemplo, suponiendo que la constante es $N = 5$, $t = (-1, 2, -3, 4, 1)$. Un programa que calcule la 'posición del máximo', al final de su ejecución, el estado que obtengamos, debe cumplir con la postcondición $t = T \wedge (\forall \gamma : 0 \leq \gamma < N : t[pmax] \geq t[\gamma]) \wedge 0 \leq pmax < N$. El primer término indica que la variable t no cambia su contenido, sigue siendo cierto (para el ejemplo)

que $t = (-1, 2, -3, 4, 1)$ en el estado final. ¿Puede ser $pmax = 2$ en el estado final?. Como debe cumplirse, $\forall \gamma : 0 \leq \gamma < N : t[pmax] \geq t[\gamma]$, vemos que $t[2] = -3$ y que $t[2] \geq t[3]$ es falso, por lo que dicho predicado es falso. ¿Puede ser $pmax = 8$?. En el ejemplo $N = 5$. La condición $0 \leq pmax < N$ no se cumple para $pmax = 8$. El único valor que cumple ambos predicados es $pmax = 3$, ya que a simple vista contiene el mayor valor. Para el estudiante, en cuanto observardor omnisciente, resulta obvio el resultado. Para un programa, formado por sentencias que se ejecutan en un ordenador, no le resulta obvio... en la ejecución, el programa debe asegurar que $t[pmax] \geq t[0] \wedge t[pmax] \geq t[1] \wedge .... \wedge t[pmax] \geq t[N-1]$ (esta es la expansión del predicado $\forall \gamma : 0 \leq \gamma < N : t[pmax] \geq t[\gamma]$). □

Ejercicio 1 *Proponga especificaciones para los siguientes programas:*

1. *Un programa que decida si una tabla de N elementos enteros está ordenada o no.*
2. *Un programa que decida si todos los elementos de una tabla de N elementos enteros son distintos o no.*

En ambos casos, presente ejemplos de posibles ejecuciones de dichos programas siguiendo el enfoque operacional de las especificaciones dado en esta sección.

2. Verificación y derivación

Sea una especificación $\{P\}$-$\{Q\}$ para una declaración de variables dada, con precondición, el predicado P, y postcondición, el predicado Q.

> Un programa S **satisface** la especificación $\{P\}$-$\{Q\}$ cuando *toda ejecución de S que commience en un estado que cumple P (i) termina; y (ii) termina finalmente en un estado que cumple Q.*

Entonces, diremos que, $\{P\}\ S\ \{Q\}$ **se cumple** *cuando, efectivamente, S satisface la especificación* $\{P\}$-$\{Q\}$.

Según lo indicado, podemos definir varios problemas. Problemas que nos encontraremos a lo largo de la Programación:

> **Problema de verificación**. Conocemos la especificación $\{P\}$-$\{Q\}$, tenemos un programa S que hemos diseñado para dicha especificación y queremos comprobar que $\{P\}\ S\ \{Q\}$ se cumple o no se cumple.

> **Cálculo de la precondición más débil**. Conocemos el programa S y los resultados que queremos obtener según una postcondición Q. Queremos calcular el predicado más débil WP, que haga que $\{WP\}\ S\ \{Q\}$ se cumpla. Si encontramos WP, como es el predicado más dédil, cualquier otro predicado que haga cierto $[P \Rightarrow WP]$, nos permite concluir que $\{P\}\ S\ \{Q\}$ se cumple también.

> **Derivación de un programa**. Conocemos símplemente la especificación del problema que queremos resolver $\{P\}$-$\{Q\}$. Queremos diseñar un programa S que, efectivamente, satisfaga dicha especificación: $\{P\}\ S\ \{Q\}$ tiene que cumplirse.

3. Reglas generales de programas

Hemos indicado anteriormente que un programa S cumple $\{P\}\ S\ \{Q\}$ cuando toda ejecución de S que comience en un estado inicial σ_i que cumple P ($\sigma_i \in \{P\}$), el programa termina, y termina en un estado final σ_f que cumple Q ($\sigma_f \in \{Q\}$).

Con esta interpretación operacional, podemos fácilmente obtener las siguientes reglas generales para el programa S que satisface la especificación $\{P\}$-$\{Q\}$:

1. Primera regla de consecuencia (reforzamiento de la precondición):

$$\{P\}\, S\, \{Q\} \text{ y } [[R \Rightarrow P]] \text{ implica } \{R\}\, S\, \{Q\} \tag{1}$$

2. Segunda regla de consecuencia (debilitamiento de la postcondición):

$$\{P\}\, S\, \{Q\} \text{ y } [[Q \Rightarrow L]] \text{ implica } \{P\}\, S\, \{L\} \tag{2}$$

Si se cumple $\{P\}\, S\, \{Q\}$ y $[[R \Rightarrow P]]$ entonces sabemos que $[[R \Rightarrow P]]$ es equivalente a que $\{R\} \subseteq \{P\}$. Por lo tanto, si comenzamos en un estado $\sigma_i \in \{R\}$, $\sigma_i \in \{P\}$ y el programa termina en un estado que cumple Q. Luego es correcto, $\{R\}\, S\, \{Q\}$. La primera regla (1) se cumple. Para la segunda regla (2), el razonamiento es muy parecido. Si $[[Q \Rightarrow L]]$ se cumple, $\{Q\} \subseteq \{L\}$. Al terminar el programa S en un estado que cumple Q, también cumple L.

Diálogo:

> -Bob: Siempre se cumple [[falso $\Rightarrow P$]]. Luego cualquier programa que cumple $\{P\}\, S\, \{Q\}$, por la primera regla cumple también $\{\text{falso}\}\, S\, \{Q\}$. Pero el conjunto {falso} está vacío. No hay estado que haga cierto al predicado falso.
> -Alice: Tienes razón Bob. Pero si comienzas en 'ningún estado', el programa S termina en ningún estado, es decir su conjunto de estado finales está vacío, $\emptyset$, y éste está contenido en $\{Q\}$, $\emptyset \subseteq \{Q\}$. En otras palabras, no encuentras que el estado final haga falso el predicado Q. Así que se cumple.
> -Bob: Entonces, un programa que no hace nada, siempre cumple la especificación.
> -Alice: Si. $\{\text{falso}\}\, S\, \{Q\}$ es siempre correcto para cualquier programa S y postcondición Q.
> -Bob: No tiene sentido escribir un programa para no ejecutarlo.
> -Alice: Eso también. No tiene mucho sentido.
>
> Comentario: Bob no tiene razón. Todo programa puede ser ejecutado desde un estado inicial. Una especificación siempre incluye una declaración de variables, y un estado está formado por los valores dados a las variables. El conjunto de todos los estados posibles no está vacío. Así que, siempre puedes ejecutar un programa. Otra cosa es, si el estado inicial que eliges cumple o no la precondición P de su especificación. La conclusión de Alice es correcta pero su razonamiento no es muy adecuado. Algunas demostraciones se hacen de forma más sencilla por lo que se llama 'reducción al absurdo'. Queremos demostrar que $\{\text{falso}\}\, S\, \{Q\}$ se cumple para cualquier programa S y postcondición Q. Supongamos que $\{\text{falso}\}\, S\, \{Q\}$ no se cumple para un programa S y Q. Por la definición operacional, eso significa que hay un estado $\sigma_i \in \{\text{falso}\}$ tal que, o bien S no termina, o bien, termina en un estado $\sigma_f \notin \{Q\}$. Pero todo estado cumple $\sigma \notin \{\text{falso}\}$. Por lo tanto, la suposición de partida es falsa (aquí está la reducción al absurdo) y, por tanto, $\{\text{falso}\}\, S\, \{Q\}$ se cumple para cualquier programa S y postcondición Q.

Ejercicio 2 *Realice los siguientes ejercicios:*

1. *Razona qué significado tiene que $\{P\}\, S\, \{cierto\}$ se cumpla para un programa S y una precondición P.*

2. *Demuestra que '$\{P\}S\{false\}$ se cumple' es equivalente a que* $[[P \equiv false]]$.
3. *Demuestra que* $\sqrt{2}$ *no es un número racional.*
4. *Dada una función* $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$*, dada una constante* C*, se denomina 'curva de nivel de valor* C*' a los puntos* $\langle x, y\rangle$ *que cumplen* $f(x, y) = C$*. Demuestra que dos curvas de nivel diferentes no tienen puntos en común, es decir, no se cortan.*

Si suponemos que tenemos un programa S que cumple $\{P\}\, S\, \{Q\}$ y que cumple $\{P\}\, S\, \{L\}$, entonces la ejecución de S comenzando en cualquier estado que cumple P termina en un estado que cumple a la vez Q y L, y por tanto, se cumple $Q \wedge L$. Por otro lado, si tenemos un programa S que cumple $\{P\}\, S\, \{Q\}$ y que cumple $\{R\}\, S\, \{Q\}$, al comenzar en un estado que cumple P o R, por tanto, un estado que cumple $P \vee R$, termina en un estado que cumple Q. Lo anterior justifica las siguientes reglas:

1. Regla de conjunción:

$$\{P\}\, S\, \{Q\} \text{ y } \{P\}\, S\, \{L\} \text{ implica } \{P\}\, S\, \{Q \wedge L\} \tag{3}$$

2. Regla de disyunción:

$$\{P\}\, S\, \{Q\} \text{ y } \{R\}\, S\, \{Q\} \text{ implica } \{P \vee R\}\, S\, \{Q\} \tag{4}$$

Ejercicio 3 *Empleando las reglas dadas en esta sección, deduce lo siguiente:*

Si $\{P_0\}\, S\, \{Q_0\}$ *y* $\{P_1\}\, S\, \{Q_1\}$ *se cumplen, entonces, el programa* S *también cumple:*
$\{P_0 \wedge P_1\}\, S\, \{Q_0 \wedge Q_1\}$ *y* $\{P_0 \vee P_1\}\, S\, \{Q_0 \vee Q_1\}$

4. Composiciones de sentencias y reglas de inferencia

Un programa es un 'texto' que contiene al menos (en programación imperativa): (1) una declaración de variables, y (2) sentencias que indican cómo se actúa sobre dichas variables. En los ejemplos de los programas explicados en la Lectura[1], hemos introducido algunas de las sentencias que vamos a utilizar a lo largo del curso. Se puede razonar sobre la corrección de las sentencias de una forma general utilizando los argumentos del enfoque operacional que hemos visto en la secciones anteriores. La forma de razonar sobre la corrección de los programas se realiza a través de lo que denominamos **reglas de inferencia**. Una regla de inferencia nos sirve para demostrar si una determinada sentencia satisace una especificación dada. Como vamos a ver a continuación una sentencia puede estar **compuesta** a su vez de otras sentencias, y la forma de su composición determina una estructura en la ejecución de los programas. Los programas entonces son básicamente sentencias que se componen de una determinada manera. Los programas que consideramos en este curso son programas **secuenciales y deterministas**: las sentencias que 'finalmente' se han ejecutado siguen (1) una única línea de ejecución, y (2) la siguiente sentencia a ejecutar se selecciona de forma determinista a partir del estado anterior a la ejecución de dicha sentencia (no hay posibilidad de una elección múltiple entre varias sentencias). En lo que presentamos a continuación asumimos que se ha hecho una declaración de variables previa y que las sentencias usan dichas variables.

4.1. Regla de inferencia de la sentencia de asignación

Dada una sentencia de asignación de la forma (genérica), $x := Exp$, y dada una especificación $\{P\}$-$\{Q\}$, entonces $\{P\}\ x := Exp\ \{Q\}$ se cumple cuando es cierto que $[[P \Rightarrow Q(x := Exp) \wedge def(Exp)]]$. La regla de inferencia queda de la siguiente manera,

Regla de inferencia de la sentencia de asignación:

$$\begin{array}{c} \{P\}\, x := Exp\, \{Q\} \text{ es equivalente a} \\ [[\, P \Rightarrow Q(x := Exp) \wedge def(Exp)\,]] \end{array} \tag{5}$$

Por el Axioma de la asignación (ver Lectura[3], sección 5) $\{Q(x := Exp) \wedge def(Exp)\}\ x := Exp\ \{Q\}$ siempre se cumple. Luego por la 'primera regla de consecuencia de las especificaciones', si $[[P \Rightarrow Q(x := Exp) \wedge def(Exp)]]$ se cumple, entonces $\{P\}\ x := Exp\ \{Q\}$ se cumple. En el otro sentido. Supongamos que $\{P\}\ x := Exp\ \{Q\}$ se cumple. Entonces la ejecución de $x := Exp$ termina. Luego, está bien definida la evaluación de la expresión Exp, $def(Exp)$ se cumple. Supongamos entonces que comenzamos en un estado $\sigma_i \in \{P\}$ pero que no cumple $Q(x := Exp)(\sigma_i)$, es decir, la evaluación de la expresión en σ_i (el valor obtenido) no cumple las condiciones de x en Q, por lo que, al ejecutarse, el estado final σ_f alcanzado tampoco cumple $Q(\sigma_f)$. Esto es una contradicción ya que $\{P\}\ x := Exp\ \{Q\}$ se cumplía. Por lo tanto, si $\sigma \in \{P\}$ entonces $\sigma \in \{Q(x := Exp) \wedge def(Exp)\}$, lo que lleva a que $[[\, P \Rightarrow Q(x := Exp) \wedge def(Exp)\,]]$ se cumple.

Por lo indicado anteriormente, hemos demostrado la regla de inferencia de la sentencia de asignación a partir del axioma de la asignación. También queda demostrado que el predicado $Q(x := Exp) \wedge def(Exp)$ es el predicado más débil que cumple $\{WP\}\ x := Exp\ \{Q\}$.

Ejercicio 4 *Obtenga la regla de inferencia de la instrucción* **continuar**.

4.2. Composición secuencial de sentencias

Sean $S1$ y $S2$ dos sentencias dadas. Para indicar en el texto del programa que $S1$ se debe ejecutar antes que $S2$ convenimos en utilizar el símbolo 'punto y coma'. Así, $S1; S2$ significa que primero ejecutamos $S1$ y a continuación ejecutamos $S2$.

La descripción **operacional** de la composción secuencial $S1; S2$ es como sigue: a partir de un estado inicial σ_i, se ejecuta $S1$, si termina, se obtiene un estado intermedio σ', a partir de este estado se ejecuta $S2$, y, si termina, se obtiene el estado final σ_f. Es posible que la ejecución de $S1$ o $S2$ falle y produzca un error abortando la ejecución.

Es importante que los programas sean correctos. Entonces, dada una especificación $\{P\}-\{Q\}$ diremos que, $\{P\}\, S1;\, S2\, \{Q\}$ se cumple si y sólo si hay un predicado R tal que $\{P\}\, S1\, \{R\}$ y $\{R\}\, S2\, \{Q\}$ se cumplen.

Regla de inferencia de la composición secuencial:

$$\begin{array}{c} \{P\}\, S1;\, S2\, \{Q\} \text{ es equivalente a} \\ \text{que exista } R \text{ tal que } \{P\}\, S1\, \{R\} \text{ y } \{R\}\, S2\, \{Q\} \end{array} \tag{6}$$

El significado operacional y la regla de inferencia para varias sentencias compuestas secuncialmente, por ejemplo, $S1;\ S2;\ S3$, es similar a lo indicado anteriormente. Observa que, en general, no es lo mismo $S1; S2$ que $S2; S1$. Los efectos secuenciales no son commutativos pero si asociativos.

Ejercicio 5 *Verifica, utilizando la regla de inferencia de la composición secuencial, el programa de intercambio de valores entre dos variables (otra cuarta forma de hacer el intercambio).*

1: ***var***
2: x, y: *entero*
3: ***fvar***
4: $\{x = A \wedge y = B\}$
5: $x := x + y;$
6: $y := x - y;$
7: $x := x - y$
8: $\{x = B \wedge y = A\}$

Ayuda: los predicados intermedios (R) se obtienen usando el axioma de la sentencias de asignación.

Ejercicio 6 *Las reglas de inferencia se pueden utilizar para verificar un programa dado, como en el ejercicio anterior. Más interesante es utilizar las reglas para calcular las sentencias que se necesitan para que se cumpla la especificación. Calcula las expresiones para que los programas siguientes cumplan su especificación:*

1.
1: ***var***
2: q, r, y: *entero*
3: ***fvar***
4: $\{A = q \cdot y + r \wedge y = B\}$
5: $q := Exp;$ ▷ *calcula Exp*
6: $r := r - y$
7: $\{A = q \cdot y + r \wedge y = B\}$

2.
1: ***var***
2: x, y: *entero*
3: ***fvar***
4: $\{cierto\}$
5: $y := Exp;$ ▷ *calcula Exp*
6: $x := x \ div \ 2$
7: $\{2 \cdot x = y\}$

3.
1: ***var***
2: x, y, p, q: *entero*
3: ***fvar***
4: $\{x \cdot y + p \cdot q = N\}$
5: $x := x - p;$
6: $q := Exp$ ▷ *calcula Exp*
7: $\{x \cdot y + p \cdot q = N\}$

4.3. Composición alternativa de sentencias

Sean $S1$ y $S2$ dos sentencias dadas. En algunos programa queremos formar una nueva sentencia que, dependiendo del estado inicial σ_i previo a su ejecución, se elija la ejecución de $S1$ o se elija la ejecución de $S2$. La condición para la elección es un predicado B tal que si $B(\sigma_i) \equiv$ cierto se elija $S1$ y si $B(\sigma_i) \equiv$ falso se elija $S2$. Este predicado B 'vigila' el camino que debe seguir la ejecución de la sentencia. Para indicar en el texto del programa que estamos formando una sentencia alternativa (con dos casos exclusivos) utilizamos la siguiente sintáxis:

1: **si** $B \to S1$
2: $[]\ \neg B \to S2$
3: **fsi**

La interpretación **operacional** de la sentencia alternativa es la siguiente: a partir de un estado σ_i, cuando se alcanza la línea del **si**, se evalúa B, se calcula $B(\sigma_i)$, si no se produce error se continúa. Si se cumple B en σ_i, se ejecuta la sentencia $S1$ a partir de σ_i y se termina en la línea **fsi**, y si no se cumple B (se cumple $\neg B$) se ejecuta la sentencia $S2$ y se termina en la línea **fsi**. Observa que la evaluación de B no modifica los valores de las variables.

Claramente la corrección de la sentencia dependerá de si son o no correctos $S1$ y $S2$. Para una especificación dada $\{P\} - \{Q\}$ la regla de inferencia es como sigue,

Regla de inferencia de la composición alternativa (2 vigilantes):

$$\begin{array}{l} \{P\}\,\textbf{si}\ B \to S1\,[]\,\neg B \to S2\ \textbf{fsi}\,\{Q\} \text{ es equivalente a} \\ \\ \text{(i) } [[P \Rightarrow def(B)]] \text{ y} \\ \text{(ii) } \{P \wedge B\}\,S1\,\{Q\} \text{ y } \{P \wedge \neg B\}\,S2\,\{Q\} \end{array} \tag{7}$$

Ejercicio 7 *Comprueba si el siguiente programa satisface su especificación. (a) Primero, recuerda el funcionamiento de la sentencia de asignación, y después, ejecuta el programa con varios ejemplos. (b) Utiliza la regla de inferencia para verificar el programa. (c) Calcula cuál es el predicado más débil para el programa dado.*

1: ***var***
2: x, y, z: *entero*
3: ***fvar***
4: $\{x = A \wedge y = B\}$
5: **si** $x \geq y \to z := x$
6: $[]\ x < y \to z := y$
7: **fsi**
8: $\{z = max(x, y) \wedge x = A \wedge y = B\}$

Ayuda: $[[\, z = max(x, y) \equiv (z = x \vee z = y) \wedge z \geq x \wedge z \geq y \,]]$. □

Ejercicio 8 *Diseña un programa mediante una composición alternativa para que cumpla la siguiente especificación,*

1: ***var***
2: a, b, u: *entero*
3: ***fvar***
4: $\{a = A \wedge b = B \wedge C = u + ab\}$
5: ▷ *diseña el programa*
6: $\{a = A\ div\ 2 \wedge b = 2B \wedge C = u + ab\}$

Ejercicio 9 *Generaliza la composición alternativa para el caso de tres vigilantes y tres sentencias $S1$, $S2$ y $S3$ que deben elegirse de manera excluyente. Dé una regla de inferencia para este caso.*

4.4. Composición iterativa de sentencias

Los programas basados en una composición iterativa, coloquialmente 'bucle', al ejecutarse, producen un proceso iterativo hasta que la condición de salida del bucle se cumple y se termina su ejecución. Hemos visto varios ejemplos en la Lectura[1], el cálculo de la raíz cuadrada, el cálculo de máximo común divisor, y en la Lectura[2], el cálculo de una matriz transpuesta de una matriz dada. En el diseño de programas iterativos es conveniente seguir una estructura para la construcción del programa. La estructura que seguiremos es la siguiente:

```
1: var
2:     variables                                  ▷ declaración de variables
3: fvar
4: {P}
5: S_inicio                                       ▷ sentencias de inicio
6: mientras B hacer                               ▷ B es la condición de continuación
7:     S_cuerpo                                   ▷ sentencias del cuerpo del bucle
8: fmientras
9: {Q}
```

Tal y como indica la estructura anterior, una vez que se ejecutan las **sentencias de inicio** S_{inicio}, se evalua la **condición de continuación** B. Si ésta se evalua sin error (en caso contrario el bucle falla) y es cierta, se entra dentro del cuerpo del bucle y se ejecutan las **sentecias del cuerpo** S_{cuerpo}, cuando se termina la ejecución de éstas, se vuelve de nuevo a comprobar la condición de continuación en la línea 6. Se realiza el mismo proceso hasta que la condición de continuación y se hace falsa. Entonces decimos que se cumple la **condición de salida** del bucle, $\neg B$, y ya no se entra dentro del bucle y se termina en la línea 8. Una vez terminado el bucle, se continúa la ejecución si hubiese más sentencias o recoges los resultados. En la estructura anterior si alguna sentencia falla todo el bucle falla. Si una expresión, en una sentencia, no se puede evaluar se produce un error. Es muy importante que el bucle termine, porque en caso contrario, estarías en un 'bucle infinito'. Este error es muy común si no se tiene cuidado en el diseño del programa iterativo. Por otro lado, las sentencias de inicio o las del cuerpo de bucle pueden ser cualesquiera, sentencias de asignación, composiciones alternativas u otros bucles.

¿Cuál es la idea para verificar que el programa anterior, basado en un bucle, cumple con la especificación $\{P\} - \{Q\}$? La idea es definir un predicado I que nos permita demostrar, por partes, la verificación de todo el programa si el bucle termina!!; y luego comprobar que efectivamente el bucle termina!!. Son dos partes que hay que diferenciar. ¿Dónde ponemos el conjunto de estados que cumplen el predicado I? justo al comienzo del bucle:

```
1: var
2:     variables                                  ▷ declaración de variables
3: fvar
4: {P}
5: S_inicio                                       ▷ sentencias de inicio
6: {I}
7: mientras B hacer                               ▷ B es la condición de continuación
8:     {I ∧ B}
9:     S_cuerpo                                   ▷ sentencias del cuerpo del bucle
10:    {I}
11: fmientras
```

12: $\{I \wedge \neg B\}$
13: $\{Q\}$

El programa iterativo dado cumple parcialmente con la especificación $\{P\} - \{Q\}$ si **existe un predicado** I tal que

1. $\{P\}\ S_{inicio}\ \{I\}$ se cumple.
2. $[[\, I \Rightarrow def(B)\,]]$ es cierto.
3. $\{I \wedge B\}\ S_{cuerpo}\ \{I\}$ se cumple.
4. $[[\, I \wedge \neg B \Rightarrow Q\,]]$ se cumple, es decir que $\{I \wedge \neg B\} \subseteq \{Q\}$.

Las reglas anteriores pasan por establecer el predicado I. Si este predicado existe y se cumplen las reglas enumeradas anteriormente entonces decimos que el predicado I es **invariante**. El invariante es la 'explicación' de que el bucle, si termina, alcanza un estado final que cumple la postcondición Q. Ahora bien, ¿cómo podemos demostrar que el bucle, efectivamente, termina?.

Si un bucle termina necesitamos algún método de demostración de este hecho, por ejemplo, encontrando una contradicción sobre una ejecución que nunca termine. Supongamos que existe una función definida sobre las variables del programa a los números naturales, $f : \Gamma \to \mathbb{N}$, y que cumpla (i) antes de entrar a ejecutar el cuerpo del bucle, la función toma un valor positivo; y (ii) cada vez que ejecutas el cuerpo del bucle la función toma un valor más pequeño (decrece) que el valor que toma justa antes de ejecutar dichas sentencias. Si tal función existe entonces no puede haber una ejecución infinita del bucle, puesto que una función decreciente y positiva llegará un momento en que deje de ser positiva, lo cual es una contradicción con el hecho de que la ejecución es infinita.

Dado el programa iterativo dado y demostrado que existe un predicado invariante I, entonces el programa termina si existe una función $f : \Gamma \to \mathbb{N}$ tal que

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

Si existe tal función $f()$, entonces el bucle termina, y a dicha función la denominamos **función de cota** del bucle.

5. Ejercicios

Las demostraciones de proposiciones del tipo $[[A \wedge B \Rightarrow C \wedge B]]$ es conveniente hacerlas de manera separada. Se comprueba, por ejemplo, que $[[A \Rightarrow C]]$ y que $[[B \Rightarrow D]]$ para concluir que $[[A \wedge B \Rightarrow C \wedge B]]$. El ejemplo que se da a continuación explica con detalle el proceso de verificación de un programa iterativo.

Ejemplo 4

Dado el siguiente algoritmo comprobar que cumple su especificación.

1: **constante**
2: $\quad N > 0$
3: **fconstante**
4: **var**

5: t: **tabla** $[0..N-1]$ **de** entero
6: b: booleano
7: **fvar**
8: $\{P:\ t = T\}$
9: **var** k: entero **fvar**
10: $b := \text{falso};$ ▷ sentencias de inicio
11: $k := 0;$
12: $\{I:\ (b \equiv (\exists \gamma : 0 \leq \gamma < k : t[\gamma] = 0)) \wedge 0 \leq k \leq N \wedge t = T\}$
13: **mientras** $k \neq N$ **hacer** ▷ $k \neq N$ es la condición de continuación
14: $b := b \vee t[k] = 0;$ ▷ sentencias del cuerpo del bucle
15: $k := k + 1$
16: **fmientras**
17: $\{Q:\ (b \equiv (\exists \gamma : 0 \leq \gamma < N : t[\gamma] = 0)) \wedge t = T\}$

Como puedes observar, el programa dado sigue la estructura indicada para la composición iterativa dada en la sección anterior. ¿Qué hace el programa?. Comienza con una tabla en la variable t que ya se supone dada $t = T$. El tamaño de la tabla es N y sus índices (posiciones) van desde 0 hasta $N-1$. La constante N que indica el tamaño de la tabla tiene que ser mayor que cero para que la tabla no sea vacía. Considera que N es 4, y la tabla inicial es $[4, 7, -1, 0]$. La postcondición Q nos indica que b será cierto si hay algún elemento de la tabla que tiene valor 0, y además al final los valores en la tabla siguen siendo los mismos que al principio $t = T$. Este es el caso del ejemplo, luego b termina con valor cierto puesto que $t[3] = 0$ (en la posición con número 3 se cumple; ojo 'no en la tercerera posición'). El programa define una variable k y si ejecutas el código observas que k va tomando los valores 0, 1,..., $N-1$ y cuando toma el valor N se cumple la condición de salida del bucle que es $k = N$ y termina. Así, que dentro de las sentencias del cuerpo del bucle k nos va 'apuntado' a las posiciones de la tabla. En el cuerpo del bucle si encontramos la condición $t[k] = 0$ a verdadero entonces b tomará el valor veradero, pero si no encontramos ningún $t[k] = 0$ que sea verdadero b valdrá falso (que es el valor de b antes de entrar en el bucle). La explicación del funcionamiento del programa se dá por medio del invariante I. Este predicado indica que en todo momento antes de entrar en el bucle, b será cierto o falso en relación a que exista o no la condición $t[\gamma] = 0$ para algún γ entre 0 y justo antes de k. Además, el valor de k no puede salirse de los valores $0 \leq k \leq N$ tal y la tabla t no cambia los valores que tiene, $t = T$, tal y como indica I. Vamos a seguir paso a paso toda la verificación de este programa según las reglas de inferencia dadas en las secciones anteriores.

1. Demostrar que $\{P\}\ S_{inicio}\ \{I\}$ se cumple.
   En nuestro programa, $\{P\}\ b := \text{false};\ k := 0;\ \{I\}$. Esto es correcto si y solo si (aplicando la regla de inferencia de la composición secuencial y de la asignación) se cumple $[[P \Rightarrow ((I)_0^k)_{falso}^b]]$.

   $((I)_0^k)_{falso}^b$
   $\equiv (falso \equiv (\exists \gamma : 0 \leq \gamma < 0 : t[\gamma] = 0)) \wedge 0 \leq 0 \leq N \wedge t = T$
   $\equiv (falso \equiv (\exists \gamma : falso : t[\gamma] = 0)) \wedge 0 \leq N \wedge t = T$ (rango del cuantificador vacío)
   $\equiv (falso \equiv falso) \wedge 0 \leq N \wedge t = T$
   $\equiv 0 \leq N \wedge t = T$
   $\Leftarrow$
   $t = T \equiv P$

   La constante N cumple $N > 0$ y esta condición se puede utilizar en cualquier predicado. Por tanto, queda demostrado.

2. Demostrar que $[[\, I \Rightarrow def(B)\,]]$ es cumple. $def(B) : def(k \neq N)$ y $[[def(k \neq N) \equiv cierto]]$. Sólo tienes que preocuparte de las definiciones de las expresiones cuando haya división por una variable o cuando en la expresión aparezcan tablas!!. En los demás casos puedes ignorar esta demostración.

3. Demostrar que $\{I \wedge B\}\ S_{cuerpo}\ \{I\}$ se cumple.
   En nuestro programa, tenemos $\{I \wedge B\}\ b := b \vee t[k] = 0;\ k := k+1\ \{I\}$. Esto es correcto si y solo si (aplicando la regla de inferencia de la composición secuencial y de la asignación) se cumple $[[I \wedge B \Rightarrow ((I)^k_{k+1})^b_{b \vee t[k]=0} \wedge def(b \vee t[k] = 0)]]$
   Observa que $[[def(b \vee t[k] = 0) \equiv 0 \leq k < N]]$ puesto que las posiciones de la tabla van desde 0 hasta $N-1$.

   $((I)^k_{k+1})^b_{b \vee t[k]=0} \wedge def(b \vee t[k] = 0)$
   $\equiv$ sustituyendo
   $(b \vee t[k] = 0 \equiv (\exists \gamma : 0 \leq \gamma < k+1 : t[\gamma] = 0)) \wedge 0 \leq k+1 \leq N \wedge 0 \leq k < N \wedge t = T$
   $\equiv$ separando el cuantificador en el último término y operando los rangos de k
   $(b \vee t[k] = 0 \equiv (\exists \gamma : 0 \leq \gamma < k : t[\gamma] = 0) \vee t[k] = 0) \wedge 0 \leq k < N \wedge t = T$
   $\Leftarrow$ por la propiedad $[[(A \equiv B) \Rightarrow (A \vee C \equiv B \vee C)]]$
   $(b \equiv (\exists \gamma : 0 \leq \gamma < k : t[\gamma] = 0)) \wedge 0 \leq k < N \wedge t = T$
   $\equiv$
   $(b \equiv (\exists \gamma : 0 \leq \gamma < k : t[\gamma] = 0)) \wedge 0 \leq k \leq N \wedge t = T \wedge k \neq N$
   $\equiv$
   $I \wedge B$

4. Demostrar que $[[\, I \wedge \neg B \Rightarrow Q\,]]$ se cumple, es decir que $\{I \wedge \neg B\} \subseteq \{Q\}$. En nuestro caso, la condición de salida es $\neg B : k = N$.

   $I \wedge k = N$
   $\equiv (b \equiv (\exists \gamma : 0 \leq \gamma < k : t[\gamma] = 0)) \wedge 0 \leq k \leq N \wedge t = T \wedge k = N$
   $\equiv (b \equiv (\exists \gamma : 0 \leq \gamma < N : t[\gamma] = 0)) \wedge 0 \leq N \wedge t = T \wedge k = N$
   $\Rightarrow$ por $[[A \wedge B \Rightarrow A]]$
   $(b \equiv (\exists \gamma : 0 \leq \gamma < N : t[\gamma] = 0)) \wedge t = T$
   $\equiv$
   Q

5. LLegados a este punto sabemos que el predicado I es efectivamente un invariante para el programa y que si el programa termina cumple su especificación. Queda por tanto demostrar que efectivamente comenzando en cualquier estado que cumple P el programa termina en un estado de Q. Si ejecutamos el programa con un ejemplo, enseguida nos damos cuenta de que la razón para su terminación es que, la variable k comienza en 0 y en cada vuelta del bucle se incrementa en 1 hasta llegar al valor de N. Desde un punto de vista operacional es suficiente pero desde el punto de vista más formal la función que acota el bucle es simplemente $f() = N - K$.

   a) Comprobamos que $[[I \wedge B \Rightarrow f() > 0]]$. Es suficiente comprobar que $[[0 \leq k \leq N \wedge k \neq N \Rightarrow N - K > 0]]$ lo cual es cierto.

   b) Comprobamos que $\{I \wedge B \wedge f() = V\}\, S_{cuerpo}\, \{f() < V\}$. En nuestro caso, $\{I \wedge B \wedge f() = V\}\ b := b \vee t[k] = 0;\ k := k+1\ \{f() < V\}$. Esto se cumple si $[[I \wedge B \wedge f() = V \Rightarrow ((f() < V)^k_{k+1})^b_{b \vee t[k]=0} \wedge def(b \vee t[k] = 0)]]$
   pero como $f() = N - K$, la única sustitución que le afecta es debida a la asignación $k := k+1$. Así nos queda

$[[I \wedge B \wedge N - k = V \Rightarrow N - k - 1 < V \wedge 0 \leq k < N]]$. Así que lo que tenemos que demostrar es básicamente,
$[[0 \leq k \leq N \wedge k \neq N \wedge N - k = V \Rightarrow N - k - 1 < V \wedge 0 \leq k < N]]$.

$$\begin{array}{l} 0 \leq k \leq N \wedge k \neq N \wedge N - k = V \\ \equiv \\ 0 \leq k < N \wedge N - k - 1 = V - 1 \\ \Rightarrow \\ 0 \leq k < N \wedge N - k - 1 < V \end{array}$$

Una reflexión final: El programa anterior es muy poco eficiente para decidir si hay un elemento con valor cero en la tabla ya que obliga a recorrer todos los elementos y debería terminar justo cuando encuentra que la condición $t[k] = 0$ se cumple. Este problema es un problema de búsqueda de una propiedad en la tabla y deberemos utilizar una solución mejor. Simplemente nos ha servido para usar todos los elementos de la lógica y reglas de inferencia para su demostración. □

Ejercicio 10 *Dado un número positivo N, el factorial de dicho número, denotado $N!$, se calcula como el producto $N! = N \times (N-1) \times (N-2)... \times 1$. Según esta definición, $N! = N \times (N-1)!$ y $1! = 1$. Verifica el siguiente programa iterativo para calcular el factorial de un número positivo dado. Da una función de cota para asegurar que el programa termina. Antes de hacer la verificación haz unos ejemplos de ejecución, por ejemplo, para $N = 4$, y comprueba, que en cada estado, justo antes de comenzar el bucle, se cumple el predicado (invariante) I en dicho estado.*

```
1:  var
2:      fact, n: entero
3:  fvar
4:  {P : n = N ∧ n > 0}
5:  var i: entero fvar
6:  fact := 1;                                  ▷ sentencias de inicio
7:  i := 1;
8:  {I : fact = i! ∧ n = N ∧ 1 ≤ i ≤ n}
9:  mientras i ≠ n hacer                        ▷ i ≠ n es la condición de continuación
10:     fact := fact * (i + 1);                 ▷ sentencias del cuerpo del bucle
11:     i := i + 1
12: fmientras
13: {Q : fact = N! ∧ n = N}
```

Ejercicio 11 *Verifica el siguiente programa iterativo para calcular la suma de las N primeras potencias de dos. El programa está incompleto pero en el momento de aplicar las reglas para la verificación del cuerpo del bucle comprobarás que puedes usarlas para calcular las sentencias que faltan. Da una función de cota para asegurar que el programa termina. Al final, comprueba con un ejemplo de ejecución que el programa funciona correctamente, y comprueba también que el predicado I se cumple en cada estado de ejecución antes de comenzar el bucle.*

```
1:  var
2:      s, n: entero
3:  fvar
4:  {P : n = N ∧ n > 0}
5:  var p, i: entero fvar
6:  s := 1;                                     ▷ sentencias de inicio
```

7: $p := 1;$
8: $i := 0;$
9: $\{I : s = \sum_{\gamma=0}^{i} 2^\gamma \wedge p = 2^i \wedge 0 \leq i \leq n \wedge n = N\}$
10: ***mientras*** $i \neq n$ ***hacer*** ▷ $i \neq n$ *es la condición de continuación*
11: $\quad p := Exp1;$ ▷ *calcula* $Exp1$
12: $\quad s := Exp2;$ ▷ *calcula* $Exp2$
13: $\quad i := i + 1$
14: ***fmientras***
15: $\{Q : s = \sum_{\gamma=0}^{n} 2^\gamma \wedge n = N\}$

Indicaciones: La importancia de disponer de buenas especificaciones es crucial para el trabajo de los programadores. Por una parte, una especificación indica las **obligaciones** que debe seguir el diseñador del programa. Un programador debe diseñar sus programas para que se cumplan, jusamente, los requisitos de la especificación (su precondición para los datos de entrada y su postcondición para los resultados que obtiene). Por otro lado, para el 'usuario' de un programa conocer la especificación le sirve para saber cuáles son las condiciones que tienen que cumplir los datos al comienzo de la ejecución para que los resultados de la ejecución sean los correctos (siempre que el programa se haya diseñado correctamente para satisfacer la especificación). Desde el punto de vista de los estudiantes de Programación: antes de comenzar cualquier ejercicio de diseño de un programa o antes de comenzar a codificar un programa en un lenguaje de programación para ejecutarlo en un ordenador se **debe** comprender muy bien la especificación del programa. Es conveniente poner ejemplos de cómo funciona la especificación: para estos datos de entrada que cumplen la precondición ¿qué resultados vamos a obtener según la postcondición?. En el caso de que los predicados dados contengan cuantificadores (como ∀..., ∃..., u otros que veremos más adelante), los ejemplos sirven para comprender bien dichos predicados.

Referencias

[1] Anne Kaldewaij. Programming: The derivation of algorithms. Prentice Hall International, series in computer science. 1990.

[2] Ricardo Peña. Diseño de Programas. Formalismo y abstracción. Pearson, Prentice-Hall. 2005