Capítulo 14

Definición de un lenguaje de programación imperativo

Índice del Capítulo

En este capítulo veremos una serie de sentencias o comandos que definen un lenguaje de programación imperativo básico. Las sentencias serán definidas utilizando tripletas de Hoare. Para cada sentencia S y predicado Q, describiremos primero el resultado deseado de la ejecución de S, mediante el predicado $wp.S.Q$ (el cual representa el conjunto de todos los estados, tal que, si S se ejecuta en alguno de ellos su ejecución termina en un estado que satisface Q).

14.1. La sentencia skip

La primera sentencia que veremos es la sentencia **skip**. La ejecución de **skip** no produce cambios sobre los estados de las variables. La utilidad de este comando se entenderá más adelante, pero podemos adelantar que será como una sentencia neutra para los programas.

Queremos caracterizar la sentencia **skip** usando tripletas de Hoare. Comencemos estudiando $wp.\mathbf{skip}.Q$. Claramente el predicado más débil P tal que $\{P\}$ **skip** $\{Q\}$ es válida es Q, pues lo mínimo necesario para asegurar el estado final Q cuando no se hace nada es tener Q como estado inicial. Por lo tanto,

$$wp.\mathbf{skip}.Q \equiv Q$$

Esto podría inducirnos a caracterizar **skip** usando la tripleta $\{Q\}$ **skip** $\{Q\}$ como axioma, pero como ya hemos visto que es posible fortalecer la precondición, daremos una caracterización más general de este comando:

(14.1) Axioma.

$$\{P\}\ \mathbf{skip}\ \{Q\} \equiv P \Rightarrow Q$$

Por ejemplo, la tripleta $\{x \geq 1\}$ **skip** $\{x \geq 0\}$ es válida pues $x \geq 1 \Rightarrow x \geq 0$.

14.2. La sentencia abort

La sentencia **abort** está especificada por:

(14.2) Axioma.

$$\{P\}\ \mathbf{abort}\ \{Q\} \quad \equiv \quad P \equiv \mathit{false}$$

Después de esta regla, ¿cómo se ejecuta **abort**?. La expresión de arriba establece que este comando *nunca* debe ejecutarse, ya que solo puede hacerse en un estado que satisface *false*, y no existe tal estado. Si la ejecución de un programa alcanza alguna vez un punto en el que **abort** deba ejecutarse, entonces el programa completo es un error, y termina prematuramente.

En términos de la precondición más débil, tenemos que

$$wp.\mathbf{abort}.Q \equiv \mathit{false}$$

14.3. La sentencia de asignación

Cualquier cambio de estado que ocurra durante la ejecución de un programa se debe a la sentencia de asignación. Una sentencia de asignación tiene la forma $x := E$, donde x denota una variable o varias variables del programa separadas con coma y E denota una de expresión o varias expresiones, también separadas con coma (donde la cantidad de variables es igual a la de expresiones).

Dado un predicado Q, lo mínimo que se requiere para que Q sea verdadera luego de ejecutar $x := E$ es el predicado $Q[x := E]$. Es decir,

$$wp.(x := E).Q \equiv Q[x := E]$$

Entonces, para que la terna $\{P\}\ x := E\ \{Q\}$ sea correcta, el predicado P debe ser más fuerte que $Q[x := E]$. De esta manera, la asignación está especificada por:

(14.3) Axioma.

$$\{P\}\ x := E\ \{Q\}\ \equiv\ P \Rightarrow Q[x := E]$$

Asumiremos que en E solo se admitirán expresiones bien definidas, sino habría que agregar un predicado $def.E$ (que sea verdadero solo si E está bien definida) en el consecuente de la implicación: $\{P\}\ x := E\ \{Q\}\ \equiv\ P \Rightarrow def.E \wedge Q[x := E]$.

Notemos que el símobolo := utilizado en el programa $x := E$ no es el mismo que el usado en el predicado $Q[x := E]$, la primera expresión es una sentencia válida del lenguaje de programación que estamos definiendo, mientras que en la segunda expresión, := es una operación sintáctica que se aplica a expresiones lógicas, que llamamos sustitución.

(14.1) Ejemplo. Probaremos que la tripleta $\{x \geq 3\}\ x := x + 1\ \{x \geq 0\}$ es válida.

utilizando el axioma 14.3.

$$\begin{array}{ll} & \{x \geq 3\}\ x := x + 1\ \{x \geq 0\} \\ = & \quad \langle \text{axioma 14.3} \rangle \\ & x \geq 3 \Rightarrow (x \geq 0)[x := x + 1] \\ = & \quad \langle \text{aplicación de sustitución} \rangle \\ & x \geq 3 \Rightarrow x + 1 \geq 0 \\ = & \quad \langle \text{aritmética} \rangle \\ & x \geq 3 \Rightarrow x \geq -1 \\ = & \quad \langle \text{aritmética} \rangle \\ & x \geq 3 \Rightarrow -1 \leq x < 3 \vee x \geq 3 \\ = & \quad \langle \text{deb/fort a)} \rangle \\ & true \end{array}$$

Por lo tanto, hemos probado que el programa $x := x+1$ satisface la especificación dada.

(14.2) Ejemplo. También utilizaremos el axioma 14.3 para obtener programas correctos según la especificación. Por ejemplo, encontraremos e tal que se satisfaga $\{true\}\ x, y := x + 1, e\ \{y = x + 1\}$

Según el axioma 14.3, para que el programa sea correcto E debe cumplir:

$$\begin{array}{ll} & true \Rightarrow (y = x + 1)\,[x, y := x + 1, e] \\ = & \quad \langle \text{Elemento neutro a izquierda de } \Rightarrow \rangle \\ & (y = x + 1)\,[x, y := x + 1, e] \\ = & \quad \langle \text{aplicación de sustitución} \rangle \\ & e = x + 1 + 1 \\ = & \quad \langle \text{aritmética} \rangle \\ & e = x + 2 \end{array}$$

Es decir, cualquier expresión e que sea equivalente a $x + 2$ es válida; en particular, podemos tomar $e = x + 2$.

(14.3) Ejemplo. Por último, hallaremos un programa que satisfaga $\{x = A \wedge y = B\}\ \ x, y := e, f\ \ \{x = 2 \times B \wedge y = 2 \times A\}$.

Según 14.3, las expresiones e y f deben cumplir:

$x = A \wedge y = B \Rightarrow (x = 2 \times B \wedge y = 2 \times A)\,[x, y := e, f]$

Suponemos cierto el antedecente $x = A \wedge y = B$ y partimos del consecuente,

$$
\begin{array}{ll}
 & (x = 2 \times B \wedge y = 2 \times A)\,[x, y := e, f] \\
= & \quad \langle \text{aplicación de sustitución} \rangle \\
 & e = 2 \times B \wedge f = 2 \times A \\
= & \quad \langle \text{antecedente} \rangle \\
 & e = 2 \times y \wedge f = 2 \times x
\end{array}
$$

14.4. Concatenación o composición

La concatenación permite escribir secuencias de sentencias. La ejecución de dos sentencias S y T, una a continuación de la otra se indicará separando las mismas con punto y coma $(S\,;T)$, o una debajo de la otra, siendo la de la izquierda (o la de arriba) la primera que se ejecuta. Para demostrar $\{P\}\ \ S\,;T\ \ \{Q\}$, hay que encontrar un predicado intermedio R que sirva como poscondición de S y precondición de T, es decir:

(14.4) Axioma.

$$\{P\}\ \ S\,;T\ \ \{Q\} \equiv (\exists R :: \{P\}\ \ S\ \ \{R\} \wedge \{R\}\ \ T\ \ \{Q\})$$

La precondición más débil de la concatenación $S\,;T$ con respecto al predicado Q, se obtiene tomando primero la precondición más débil de T con respecto a Q y luego la precondición más débil de S con respecto a este último predicado. Es decir,

$$wp.(S\,;T).Q \equiv wp.S.(wp.T.Q)$$

(14.4) Ejemplo. Como ejemplo probaremos que el programa dado a continuación es correcto, probando la siguiente tripleta:

$$
\begin{array}{ll}
|[& \textbf{var}\ a, b : \textbf{Boolean}; \\
 & \{(a \equiv A) \wedge (b \equiv B)\} \\
 & a := a \equiv b \\
 & b := a \equiv b \\
 & a := a \equiv b \\
 & \{(a \equiv B) \wedge (b \equiv A)\} \\
]| &
\end{array}
$$

Para ello anotaremos el programa con los siguientes predicados intermedios:

$$
\begin{array}{ll}
|[& \textbf{var } a, b : \textbf{Boolean}; \\
& \{P : (a \equiv A) \wedge (b \equiv B)\} \\
& a := a \equiv b \\
& \{R_2 : wp.(b := a \equiv b).R_1\} \\
& b := a \equiv b \\
& \{R_1 : wp.(a := a \equiv b).Q\} \\
& a := a \equiv b \\
& \{Q : (a \equiv B) \wedge (b \equiv A)\} \\
]| &
\end{array}
$$

y probaremos que las tripletas:

$$
\begin{array}{l}
\{P\}\, a := a \equiv b\, \{R_2\} \\
\{R_2\}\, b := a \equiv b\, \{R_1\} \\
\{R_1\}\, a := a \equiv b\, \{Q\}
\end{array}
$$

son válidas. Las letras P, Q, R_1 y R_2 fueron agregadas para hacer más legible la notación.

Por el axioma de asignación 14.3 podemos concluir que las dos últimas tripletas son ciertas. Con lo cual, solo debemos probar la primera tripleta. Hallamos primero R_1 y R_2:

$$
\begin{array}{ll}
 & ((a \equiv B) \wedge (b \equiv A))[a := a \equiv b] \\
= & \quad \langle \text{aplicación de sustitución} \rangle \\
 & (a \equiv b \equiv B) \wedge (b \equiv A)
\end{array}
$$

$$
\begin{array}{ll}
 & ((a \equiv b \equiv B) \wedge (b \equiv A))[b := a \equiv b] \\
= & \quad \langle \text{aplicación de sustitución} \rangle \\
 & (a \equiv a \equiv b \equiv B) \wedge (a \equiv b \equiv A) \\
= & \quad \langle \text{elemento neutro de } \equiv \rangle \\
 & (b \equiv B) \wedge (a \equiv b \equiv A)
\end{array}
$$

La primera tripleta es cierta si:

$$
\begin{array}{ll}
 & (a \equiv A) \wedge (b \equiv B) \Rightarrow ((b \equiv B) \wedge (a \equiv b \equiv A))[a := a \equiv b] \\
= & \quad \langle \text{aplicación de sustitución} \rangle \\
 & (a \equiv A) \wedge (b \equiv B) \Rightarrow (b \equiv B) \wedge (a \equiv b \equiv b \equiv A) \\
= & \quad \langle \text{elemento neutro de } \equiv \rangle \\
 & (a \equiv A) \wedge (b \equiv B) \Rightarrow (b \equiv B) \wedge (a \equiv A) \\
= & \quad \langle \text{reflexividad de } \equiv \rangle \\
 & \mathit{true}
\end{array}
$$

14.5. La sentencia alternativa

Los lenguajes de programación imperativos usualmente proveen una notación para el comando llamado condicional o sentencia **if**, el cual permite la ejecución de sentencias dependiendo del estado de las variables del programa. Un ejemplo del comando condicional es el siguiente, escrito es Pascal:

$$\textbf{if } x \geq 0 \textbf{ then } z := x \textbf{ else } z := -x \tag{14.1}$$

La ejecución de este comando almacena el valor absoluto de x en z. Si $x \geq 0$ entonces se ejecuta la primer sentencia $z := x$, sino se ejecuta la segunda sentencia $z := -x$.

En la notación que utilizaremos en nuestro lenguaje de programación, este comando se escribe como:

$$\begin{array}{lrcl} \textbf{if} & x \geq 0 & \rightarrow & z := x \\ [] & \neg(x \geq 0) & \rightarrow & z := -x \\ \textbf{fi} & & & \end{array}$$

Este comando contiene dos expresiones de la forma $B \rightarrow S$ (separadas por el símbolo []), donde B es una expresión booleana, llamada guarda y S es una sentencia. Para ejecutar el comando se debe encontrar una guarda verdadera y ejecutar su sentencia correspondiente.

La forma general de esta sentencia es la siguiente:

$$\begin{array}{lllll} IF: & \textbf{if} & B_0 & \rightarrow & S_0 \\ & [] & B_1 & \rightarrow & S_1 \\ & \vdots & & & \\ & [] & B_n & \rightarrow & S_n \\ & \textbf{fi} & & & \end{array}$$

donde para todo $0 \leq i \leq n$, B_i es una expresión booleana y S_i es una sentencia.

La sentencia IF funciona de la siguiente manera: todas las guardas son evaluadas; si ninguna es $True$ se produce un **abort**; en caso contrario, se elige (de alguna manera) una guarda B_i que sea $True$ y se ejecuta la correspondiente sentencia S_i.

Ya hemos visto que **abort** no es una sentencia ejecutable; por lo tanto, un **if** con ninguna guarda será considerado un error

Si más de una guarda es *true*, la sentencia alternativa se dice *no determinística*. El "no-determinismo", es útil para escribir con claridad determinados programas.

Teniendo en cuenta lo dicho anteriormente podemos especificar la sentencia IF de la siguiente forma:

(14.5) Axioma.

$$
\begin{array}{llllll}
\{P\} & \textbf{if} & B_0 & \rightarrow & S_0 & \{Q\} \quad \equiv \quad (P \Rightarrow B_0 \vee B_1 \vee \ldots \vee B_n) \\
 & [] & B_1 & \rightarrow & S_1 & \qquad\quad \wedge \{P \wedge B_0\}\, S_0\, \{Q\} \\
 & \vdots & & & & \qquad\quad \wedge \{P \wedge B_1\}\, S_1\, \{Q\} \\
 & [] & B_n & \rightarrow & S_n & \qquad\quad \vdots \\
 & \textbf{fi} & & & & \qquad\quad \wedge \{P \wedge B_n\}\, S_n\, \{Q\}
\end{array}
$$

Por lo tanto, para probar $\{P\}\, IF\, \{Q\}$, es suficiente con probar:

i) $P \Rightarrow B_0 \vee B_1 \vee \ldots \vee B_n$

ii) $\{P \wedge B_i\}\, S_i\, \{Q\}$ para $0 \leq i \leq n$.

La implicación $P \Rightarrow B_0 \vee B_1 \vee \ldots \vee B_n$ es necesaria para asegurarse que si la precondición es cierta al menos una de las guardas es verdadera y por lo tanto el programa no es erróneo (no se produce un **abort**). Las expresiones $\{P \wedge B_i\}\, S_i\, \{Q\}$ aseguran que ejecutar S_i, cuando la precondición y la guarda S_i son verdaderas, conduce a un estado que satisface Q, es decir, cualquiera sea la guarda verdadera, ejecutar el respectivo comando conduce al estado final deseado.

De la misma forma que hicimos con la asignación, asumiremos que las expresiones B_i están bien definidas, de otra forma tendríamos que agregar una condición más en el término derecho de la $\equiv$.

La definición de $wp.IF.Q$ no es muy simple. Lo mínimo requerido para que Q sea cierta luego de ejecutar IF es que al menos una de las guardas sea cierta y que la ejecución de cada comando S_i con guarda B_i verdadera, termine en un estado que satisfaga Q. Esto último lo expresamos como: $B_i \Rightarrow wp.S_i.Q$.

$$wp.IF.Q = (B_0 \vee B_1 \vee \ldots \vee B_n) \wedge (B_0 \Rightarrow wp.S_0.Q) \wedge \ldots \wedge (B_n \Rightarrow wp.S_n.Q)$$

(14.5) Ejemplo.

Para probar que la siguiente tripleta es cierta,

$$
\begin{array}{l}
\{true\} \\
\qquad \textbf{if} \quad x \leq y \quad \rightarrow \quad \textbf{skip} \\
\qquad [] \quad x > y \quad \rightarrow \quad x, y := y, x \\
\qquad \textbf{fi} \\
\{x \leq y\}
\end{array}
$$

debemos probar:

i) $true \Rightarrow x \leq y \vee x > y$

ii) $\{true \wedge x \leq y\}$ **skip** $\{x \leq y\} \wedge \{true \wedge x > y\}\ x, y := y, x\ \{x \leq y\}$

La demostración de i) es inmediata, dado que $(x \leq y \vee x > y) \equiv true$. Para demostrar ii) probaremos cada una de las tripletas:

$$
\begin{array}{ll}
 & \{true \wedge x \leq y\}\ \textbf{skip}\ \{x \leq y\} \\
= & \quad \langle \text{axioma 14.1} \rangle \\
 & true \wedge x \leq y \Rightarrow x \leq y \\
= & \quad \langle \text{elemento neutro de } \wedge \text{ ; reflexividad de } \Rightarrow \rangle \\
 & true
\end{array}
$$

$$
\begin{array}{ll}
 & \{true \wedge x > y\}\ x, y := y, x\ \{x \leq y\} \\
= & \quad \langle \text{axioma 14.3} \rangle \\
 & true \wedge x > y \Rightarrow (x \leq y)[x, y := y, x] \\
= & \quad \langle \text{elemento neutro de } \wedge \text{ ; aplicación de sustitución } \rangle \\
 & x > y \Rightarrow y \leq x \\
= & \quad \langle \text{aritmética} \rangle \\
 & x > y \Rightarrow x > y \vee x = y \\
= & \quad \langle \text{deb/fort a)} \rangle \\
 & true
\end{array}
$$

(14.6) Ejemplo. Veamos otra situación, donde queremos determinar un programa que satisfaga la siguiente especificación:

$$\{true\}\ z :=?\ \{z = max(x, y)\}$$

Vamos a utilizar la siguiente definición de máximo entre dos números:

$$z = max(x, y) \equiv (z = x \vee z = y) \wedge z \geq x \wedge z \geq y$$

Esto significa que un posible valor para el máximo es el valor de x, por tanto una sentencia posible a ejecutar es $z := x$. Veamos entonces cuál es la precondición más débil para la sentencia de asignación $z := x$ respecto del predicado $z = max(x, y)$:

$$
\begin{array}{ll}
 & ((z = x \vee z = y) \wedge z \geq x \wedge z \geq y)[z := x] \\
= & \quad \langle \text{Sustitución} \rangle \\
 & (x = x \vee x = y) \wedge x \geq x \wedge x \geq y) \\
= & \quad \langle \text{def. de igualdad; } (x \geq x) \equiv true \rangle \\
 & (true \vee x = y) \wedge true \wedge x \geq y) \\
= & \quad \langle \text{elemento neutro de } \wedge \text{ ; absorbente de } \vee \rangle \\
 & x \geq y
\end{array}
$$

Lo cual demuestra que si elegimos el predicado $x \geq y$, la tripleta $\{x \geq y\}\ z := x\ \{z = max(x, y)\}$ es válida.

Con el mismo razonamiento también será válida $\{y \geq x\}\ z := y\ \{z = max(x, y)\}$.

Además, observando que

$$
\begin{array}{ll}
 & x \geq y \vee y \geq x \\
= & \quad \langle \text{aritmética} \rangle \\
 & x = y \vee x > y \vee \neg(x > y) \\
= & \quad \langle \text{ tercero excluido; elemento absorvente de } \vee \rangle \\
 & true
\end{array}
$$

es fácil ver que $true \Rightarrow x \geq y \vee y \geq x$

Hasta aquí hemos probado:

i) $true \Rightarrow x \geq y \vee y \geq x$

ii) $\{x \geq y\}\ z := x\ \{z = max(x, y)\}\ \wedge\ \{y \geq x\}\ z := y\ \{z = max(x, y)\}$

Por lo tanto hemos encontrado un programa que es correcto respecto a la especificación dada:

if $x \geq y \ \rightarrow\ z := x$
[] $y \geq x \ \rightarrow\ z := y$
fi

14.6. Repetición

La lista de comandos se completa con una sentencia que permitirá la ejecución repetida de una acción (ciclo o bucle). La sintaxis es :

$$
\begin{array}{lll}
DO: & \textbf{do} & B_0 \rightarrow S_0 \\
 & [] & B_1 \rightarrow S_1 \\
 & \vdots & \\
 & [] & B_n \rightarrow S_n \\
 & \textbf{od} &
\end{array}
$$

donde para todo $0 \leq i \leq n$, B_i es una expresión booleana (también llamada guarda) y S_i una sentencia.

El ciclo funciona de la siguinete manera: mientras exista alguna guarda equivalente a *true*, se elige alguna (de alguna manera) y se ejecuta la sentencia correspondiente; luego vuelven a evaluarse las guardas. Esto se repite hasta que ninguna guarda sea *true*. Si desde el comienzo del ciclo, ninguna guarda es *true*, el ciclo completo equivale a un *skip*.

Luego de ejecutarse el ciclo todas las guardas son falsas. Llamaremos *iteración de un ciclo* al proceso que consiste en elegir una guarda y ejecutar su sentencia correspondiente.

Esta sentencia también puede ser no-determinista; si más de una guarda es verdadera solo se elige una por iteración.

La especificación de esta sentencia es la siguiente:

(14.6) Axioma.

$$
\begin{array}{lllll}
\{P\} & \textbf{do} & B_0 \to S_0 & \{Q\} & \equiv \quad (P \wedge \neg B_0 \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \Rightarrow Q) \\
 & [] & B_1 \to S_1 & & \quad \wedge \{P \wedge B_0\}\ S_0\ \{P\} \\
 & \vdots & & & \quad \wedge \{P \wedge B_1\}\ S_1\ \{P\} \\
 & [] & B_n \to S_n & & \quad \vdots \\
 & \textbf{od} & & & \quad \wedge \{P \wedge B_n\}\ S_n\ \{P\} \\
 & & & & \quad \wedge \text{ el ciclo termina}
\end{array}
$$

Es decir que para probar que $\{P\}\ DO\ \{Q\}$ es correcta, es suficiente con probar:

i) $P \wedge \neg B_0 \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \Rightarrow Q$

ii) $\{P \wedge B_i\}\ S_i\ \{P\}$ para $0 \leq i \leq n$

iii) el ciclo termina

La implicación $P \wedge \neg B_0 \wedge \neg B_1 \wedge \ldots \wedge \neg B_n \Rightarrow Q$ asegura el cumplimiento de la post-condición cuando el ciclo termina, es decir cuando ninguna guarda es verdadera; mientras que $\{P \wedge B_i\}\ S_i\ \{P\}$ asegura que ejecutar S_i cuando se satisface $P \wedge B_i$ conduce a un estado que satisface P, lo cual es necesario para que tenga sentido volver a evaluar las guardas. La condición "el ciclo termina" es necesaria porque de otra manera el programa podría ejecutarse indefinidamente. Más adelante formalizaremos esta condición.

Asumiremos que las expresiones B_i están todas bien definidas.

Un predicado P que satisface $\{P \wedge B_i\}\ S_i\ \{P\}$ se llama *invariante*. Encontrar invariantes es la dificultad mayor a la hora de escribir bucles. Existen varias técnicas para determinar invariantes, por ahora solo veremos el funcionamiento de los ciclos suponiendo que ya se ha determinado el invariante.

(14.7) Ejemplo. Utilizaremos el axioma 14.6 para probar la siguiente tripleta de Hoare.

$\{P : 0 \leq i \leq n \wedge z = i \times x\}$
do $i \neq n \to i, z := i + 1, z + x$ **od**
$\{P \wedge i = n\}$

Este programa calcula el producto de valores de las variables x y n y lo guarda en z. El invariante propuesto para el bucle es el predicado P.

Probaremos primero el punto i),

$$
\begin{array}{ll}
 & P \ \wedge \ \neg(i \neq n) \Rightarrow P \ \wedge \ i = n \\
= & \quad \langle \text{definición de } \neq \text{ ; doble negación} \rangle \\
 & P \ \wedge \ i = n \Rightarrow P \ \wedge \ i = n \\
= & \quad \langle \text{reflexividad de } \Rightarrow \rangle \\
 & \mathit{true}
\end{array}
$$

Ahora probamos el punto ii),

$$
\begin{array}{ll}
 & \{P \ \wedge \ i \neq n\}\, i, z := i + 1, z + x \,\{P\} \\
= & \quad \langle \text{axioma 14.3} \rangle \\
 & P \ \wedge \ i \neq n \Rightarrow P[i, z := i + 1, z + x] \\
= & \quad \langle \text{definición de } P\text{; aplicación de sistitución} \rangle \\
 & 0 \leq i \leq n \ \wedge \ z = i \times x \ \wedge \ i \neq n \Rightarrow 0 \leq i + 1 \leq n \ \wedge \ z + x = (i + 1) \times x \\
= & \quad \langle \text{aritmética} \rangle \\
 & 0 \leq i \leq n - 1 \ \wedge \ z = i \times x \Rightarrow -1 \leq i \leq n - 1 \ \wedge \ z = i \times x \\
= & \quad \langle \text{aritmética} \rangle \\
 & 0 \leq i \leq n - 1 \ \wedge \ z = i \times x \Rightarrow (i = -1 \vee 0 \leq i \leq n - 1) \ \wedge \ z = i \times x \\
= & \quad \langle \text{deb/fort e)} \rangle \\
 & \mathit{true}
\end{array}
$$

La demostración formal de que el ciclo termina la veremos a continuación, antes podemos ver fácilmente que esta condición se cumple observando los valores que va tomando la variable i, ésta toma un valor no negativo antes de la ejecución del ciclo y aumenta uno en cada iteración, hasta alcanzar el valor n, con lo cual la guarda se evalúa falsa.

14.6.1. Terminación de ciclos

Para probar que un ciclo termina se debe encontrar una función $t : \mathit{Estados} \rightarrow \mathit{Int}$ que dado un estado de las variables del programa provea una cota superior del número de iteraciones que falten realizarse. Llameremos a *t función cota*. Podemos asegurarnos que t es una función cota si t decrece en una unidad en cada iteración y si a medida que se realiza una iteración la condición $t > 0$ se mantiene. Escribimos ésto formalmente de la siguiente manera:

Un ciclo termina si existe una función cota t tal que:

i) $P \wedge (B_0 \vee B_1 \vee \ldots \vee B_n) \Rightarrow t \geq 0$

ii) $\{P \wedge B_i \wedge t = T\} \ S_i \ \{t < T\}$ para $0 \leq i \leq n$.

La condición i) asegura que mientras alguna guarda es verdadera, la función cota es no negativa; o bien, leído al revés, si la función cota se hace negativa, entonces ninguna guarda es verdadera (puesto que P es el invariante y es siempre verdadero).

La condición ii) asegura que la función cota decrece luego de la ejecución de cualquier S_i. De esta manera, en algún momento la cota será negativa y por lo explicado más arriba el ciclo habrá terminado.

En el ejemplo anterior una posible elección para la función cota es $t = n - i$. Para hallar esta función tuvimos en cuenta que el programa termina cuando i alcanza el valor n y que inicialmente $i \leq n$ y aumenta uno en cada iteración.

La prueba formal de que el ciclo termina es la siguiente:

Prueba de i):

$$
\begin{array}{ll}
 & 0 \leq i \leq n \ \wedge \ z = i \times x \ \wedge \ i \neq n \\
= & \quad \langle \text{aritmética} \rangle \\
 & 0 \leq i \wedge i \leq n \ \wedge \ z = i \times x \ \wedge \ i \neq n \\
\Rightarrow & \quad \langle \text{deb/fort b)} \rangle \\
 & i \leq n \\
= & \quad \langle \text{aritmética} \rangle \\
 & n - i \geq 0
\end{array}
$$

Prueba de ii):

$$
\begin{array}{ll}
 & \{P \ \wedge \ i \neq n \ \wedge \ t = T\} \ i, z := i + 1, z + x \ \{t < T\} \\
= & \quad \langle \text{axioma 14.3} \rangle \\
 & P \ \wedge \ i \neq n \ \wedge \ t = T \Rightarrow (t < T)[i, z := i + 1, z + x] \\
= & \quad \langle \text{definición de } t\text{; sustitución} \rangle \\
 & P \ \wedge \ i \neq n \ \wedge \ n - i = T \Rightarrow n - (i + 1) < T
\end{array}
$$

Para continuar la prueba suponemos cierto el antecedente y demostramos el consecuente:

$$
\begin{array}{ll}
 & n - (i + 1) < T \\
= & \quad \langle \text{aritmética; antecedente } n - i = T \rangle \\
 & n - i - 1 < n - i \\
= & \quad \langle \text{aritmética} \rangle \\
 & \mathit{true}
\end{array}
$$

14.7. Ejemplos

En esta sección veremos algunos ejemplos de corrección de ciclos en contextos más generales, por ejemplo donde se inicializan las variables antes de comenzar el ciclo.

Las derivaciones de programas que contienen ciclos se basan en el cálculo invariantes. Por simplicidad solo haremos verificaciones de programas que contienen ciclos, en lugar de derivaciones.

(14.8) Ejemplo. Probaremos que el siguiente programa es correcto respecto a la especificación dada.

```
|[  const n : Integer;
    var s : Integer;
        b : array[0 . . . n] of Integer;
    {n ≥ 0}
    i, s := 1, b[0]
    do i ≤ n → i, s := i + 1, s + b[i] od
    {s = (∑ i : 0 ≤ k ≤ n : b[k])}
]|
```

Comenzaremos proponiendo un invariante P, es decir, un predicado que sea cierto luego de cada iteración del ciclo. Utilizaremos este predicado para anotar el programa con un predicado intermedio que sirva como postcondición de la sentencia $i, s := 1, b[0]$ y precondición del ciclo.

```
|[  const n : Integer;
    var s : Integer;
        b : array[0 . . . n] of Integer;
    {n ≥ 0}
    i, s := 1, b[0]
    {P}
    do i ≤ n → i, s := i + 1, s + b[i] od
    {s = (+ i : 0 ≤ k < n + 1 : b[k])}
]|
```

Las variables que cambian de estado en cada iteración son i y s, la primera toma valores dentro del rango $1, \ldots, n+1$ y la variables s acumula siempre la suma de los primeros i elementos del arreglo. Por lo tanto, proponemos el invariante: $1 \leq i \leq n+1 \wedge s = (+\, k : 0 \leq k < i : b[k])$.

Para probar que la tripleta es cierta utilizaremos el axioma 14.4 y probaremos que:

1) $\{n \geq 0\}\ i, s := 1, b[0]\ \{P\}$
2) $\{P\}$ **do** $i \leq n \rightarrow i, s := i + 1, s + b[i]$ **od** $\{s = (+\, i : 0 \leq k < n + 1 : b[k])\}$

El punto 1) es muy sencillo y lo dejamos como ejercicio. Para probar el punto 2) debemos demostrar los siguientes items:

i) $P \wedge \neg(i \leq n) \Rightarrow s = (+\, i : 0 \leq k \leq n : b[k])$

ii) $\{P \wedge i \leq n\}\ i, s := i + 1, s + b[i]\ \{P\}$

iii) Existe una función t tal que:

   a) $P \wedge i \leq n \Rightarrow t \geq 0$

   b) $\{P \wedge i \leq n \wedge t = T\}\ i, s := i + 1, s + b[i]\ \{t < T\}$

Prueba de i):

$$
\begin{array}{ll}
 & 1 \le i \le n+1 \wedge s = (+k : 0 \le k < i : b[k]) \wedge \neg(i \le n) \\
= & \quad \langle \text{aritmética} \rangle \\
 & 1 \le i \le n+1 \wedge s = (+k : 0 \le k < i : b[k]) \wedge i > n \\
= & \quad \langle \text{aritmética} \rangle \\
 & i = n+1 \wedge s = (+k : 0 \le k < i : b[k]) \\
\Rightarrow & \quad \langle \text{regla de sustitución; deb/fort b)} \rangle \\
 & s = (+k : 0 \le k < n+1 : b[k])
\end{array}
$$

Prueba de ii):

$$
\begin{array}{ll}
 & \{P \wedge i \le n\}\ i, s := i+1, s + b[i]\ \{P\} \\
= & \quad \langle \text{axioma 14.3} \rangle \\
 & P \wedge i \le n \Rightarrow P[i, s := i+1, s + b[i]] \\
= & \quad \langle \text{definición de } P\text{; sustitución} \rangle \\
 & 1 \le i \le n+1 \ \wedge\ s = (+k : 0 \le k < i : b[k]) \ \wedge\ i \le n \Rightarrow \\
 & 1 \le i+1 \le n+1 \ \wedge\ s + b[i] = (+k : 0 \le k < i+1 : b[k]) \\
= & \quad \langle \text{aritmética; separación de un término} \rangle \\
 & 1 \le i \le n \ \wedge\ s = (+k : 0 \le k < i : b[k]) \Rightarrow \\
 & 0 \le i \le n \ \wedge\ s + b[i] = b[i] + (+k : 0 \le k < i : b[k]) \\
= & \quad \langle \text{aritmética; separación de un término} \rangle \\
 & 1 \le i \le n \ \wedge\ s = (+k : 0 \le k < i : b[k]) \Rightarrow \\
 & (i = 0 \vee 1 \le i \le n) \ \wedge\ s = (+k : 0 \le k < i : b[k]) \\
= & \quad \langle \text{deb/fort e)} \rangle \\
 & \mathit{true}
\end{array}
$$

Para la prueba de iii) proponemos la función cota $t = n - i$, la demostración es similar a la del ejemplo anterior y la dejamos como ejercicio.

(14.9) Ejemplo. Por último veremos la corrección del siguiente ciclo que calcula el máximo común divisor entre dos números enteros positivos:

```
|[  var x, y : Integer
    {P : x > 0 ∧ y > 0 ∧ mcd.x.y = mcd.X.Y}
    do x > y → x := x − y
    [] y > x → y := y − x
    od
    {x = mcd.X.Y}
]|
```

Recordemos las siguientes propiedades de *mcd*:

1) $mcd.x.x = x$

2) $x > y \Rightarrow mcd.x.y = mcd.(x - y).y$

3) $y > x \Rightarrow mcd.x.y = mcd.x.(y - x)$

Primero probaremos que la postcondición se satisface al terminar el ciclo y que P es un invariante, es decir:

i) $P \wedge \neg(x > y) \wedge \neg(y > x) \Rightarrow x = mcd.X.Y$

ii) $\{P \wedge x > y\}\ x := x - y\ \{P\}$

iii) $\{P \wedge y > x\}\ y := y - x\ \{P\}$

Prueba de i):

$$
\begin{array}{ll}
 & P \wedge \neg(x > y) \wedge \neg(y > x) \\
= & \quad \langle \text{aritmética} \rangle \\
 & P \wedge x = y \\
= & \quad \langle \text{definición de } P\text{; regla de sustitución} \rangle \\
 & x > 0 \wedge x > 0 \wedge mcd.x.x = mcd.\mathtt{X}.\mathtt{Y} \\
\Rightarrow & \quad \langle \text{deb/fort b)} \rangle \\
 & mcd.x.x = mcd.\mathtt{X}.\mathtt{Y} \\
= & \quad \langle \text{Propiedad 1)} \rangle \\
 & x = mcd.\mathtt{X}.\mathtt{Y}
\end{array}
$$

Prueba de ii):

$$
\begin{array}{ll}
 & \{P \wedge x > y\}\ x := x - y\ \{P\} \\
= & \quad \langle \text{axioma 14.3} \rangle \\
 & P \wedge x > y \Rightarrow P[x := x - y] \\
= & \quad \langle \text{definición de } P\text{; sustitución} \rangle \\
 & x > 0 \wedge y > 0 \wedge mcd.x.y = mcd.\mathtt{X}.\mathtt{Y} \wedge x > y \Rightarrow \\
 & x - y > 0 \wedge y > 0 \wedge mcd.(x - y).y = mcd.\mathtt{X}.\mathtt{Y} \\
= & \quad \langle \text{aritmética; propiedad 2)} \rangle \\
 & x > 0 \wedge x > 0 \wedge mcd.(x - y).y = mcd.\mathtt{X}.\mathtt{Y} \wedge x > y \Rightarrow \\
 & x > y \wedge y > 0 \wedge mcd.(x - y).y = mcd.\mathtt{X}.\mathtt{Y} \\
= & \quad \langle \text{deb/fort b)} \rangle \\
 & true
\end{array}
$$

La prueba de iii) es similar a ii).

Tomaremos como función cota $t = x + y$, entonces debemos probar que t decrece en cada iteración y que mientras se ejecuta el ciclo $t > 0$, es decir:

iv) $\{P \wedge x > y \wedge t = T\}\ x := x - y\ \{t < T\}$

v) $\{P \wedge y > x \wedge t = T\}\ y := y - x\ \{t < T\}$

vi) $P \wedge x > y \Rightarrow t > 0$

vii) $P \wedge y > x \Rightarrow t > 0$

La prueba de que el ciclo termina se deja como ejercicio.

14.8. Ejercicios

14.1 Demostrar que las siguientes tripletas son válidas:

|[**var** x, y : **Integer**
$\{x > 0 \wedge y > 0\}$
a) **skip**
$\{x > 0\}$
]|

|[**var** x, y : **Integer**
$\{x > 0 \wedge y > 0\}$
b) **skip**
$\{y \geq 0\}$
]|

|[**var** x, y : **Boolean**;
$\{x \equiv y\}$
c) **skip**
$\{x \Rightarrow y\}$
]|

14.2 Demostrar que las siguientes tripletas no son válidas:

|[**var** x, y : **Integer**;
$\{x > 0 \wedge y > 0\}$
a) **skip**
$\{x = 1\}$
]|

|[**var** x, y : **Integer**;
$\{x > 0 \wedge y > 0\}$
b) **skip**
$\{y \geq x\}$
]|

|[**var** x, y : **Boolean**;
$\{x \equiv y\}$
c) **skip**
$\{x \wedge y\}$
]|

14.3 Determinar la precondición más débil P en cada caso:

a) $\{P\}\ x := x + 1\ \{x > 0\}$

b) $\{P\}\ x := x * x\ \{x > 0\}$

c) $\{P\}\ x := x + 1\ \{x = x + 1\}$

d) $\{P\}\ x := E\ \{x = E\}$

e) $\{P\}\ x, y := x + 1, y - 1\ \{x + y > 0\}$

f) $\{P\}\ x, y := y + 1, x - 1\ \{x > 0\}$

g) $\{P\}\ a := a \Rightarrow b\ \{a \vee b\}$

h) $\{P\}\ x := x + 1; x := x + 1\ \{x > 0\}$.

i) $\{P\}\ x := x \not\equiv y; y := x \not\equiv y; x := x \not\equiv y\ \{(x \equiv \mathtt{X}) \wedge (y \equiv \mathtt{Y})\}$.

j) $\{P\}\ x := x + 1; skip\ \left\{x^2 - 1 = 0\right\}$.

14.4 Determinar el predicado Q más fuerte para que el programa sea correcto en cada caso:

a) $\{x = 10\}\ x := x + 1\ \{Q\}$

b) $\{x \geq 10\}\ x := x - 10\ \{Q\}$

c) $\left\{x^2 > 45\right\}\ x := x + 1\ \{Q\}$

d) $\{0 \leq x < 10\}\ x := x^2\ \{Q\}$

14.5 Demostrar que la siguiente tripleta es válida: $\{x = A \wedge y = B\}\ x := x - y; y := x + y; x := x - y\ \{x = -B \wedge y = A\}$

14.6 Calcular la expresión e que haga válida la tripleta en cada caso:

a) $\{A = q \times b + r\}\ q := e; r := r - \mathtt{b}\ \{A = q \times b + r\}$.

b) $\{true\}\ y := e; x := x\, div\, 2\ \{2 \times x = y\}$.

c) $\{x \times y + p \times q = \mathtt{N}\}\ x := x - p; q := e\ \{x \times y + p \times q = \mathtt{N}\}$.

14.7 Recordemos que los números de Fibonacci $F.i$ están dados por $F.0 = 0$, $F.1 = 1$ y $F.n = F.(n-1) + F.(n-2)$ para $n \geq 2$. Sea $P : n > 0 \wedge a = F.n \wedge b = F.(n-1)$, calcular las expresiones e y f que hagan válida la tripleta:

$$\{P\}\ n, a, b := n + 1, e, f\ \{P\}$$

14.8 Demostrar que los siguientes programas son correctos, donde x, y : Integer y a, b : Boolean.

a)
$\{true\}$
if $x \geq 1 \rightarrow x := x + 1$
[] $x \leq 1 \rightarrow x := x - 1$
fi
$\{x \neq 1\}$

b)
$\{true\}$
if $x \geq y \rightarrow$ **skip**
[] $x \leq y \rightarrow x, y := y, x$
fi
$\{x \geq y\}$

c)
$\{true\}$
if $\neg a \vee b \rightarrow a := \neg a$
[] $a \vee \neg b \rightarrow b := \neg b$
fi
$\{a \vee b\}$

14.9 Encontrar el predicado P más débil que haga correcto el siguiente programa:

$\{P\}$
$x := x + 1$
if $x > 0 \rightarrow x := x - 1$
[] $x < 0 \rightarrow x := x + 2$
[] $x = 1 \rightarrow$ **skip**
fi
$\{x \geq 1\}$

14.10 Probar que las siguientes tripletas son equivalentes:

$\{P\}$
if $B_0 \rightarrow S_0; S$
[] $B_1 \rightarrow S_1; S$
fi
$\{Q\}$

$\{P\}$
if $B_0 \rightarrow S_0$
[] $B_1 \rightarrow S_1$
fi
S
$\{Q\}$

14.11 Suponiendo que el programa de la izquierda es correcto, demostrar que el de la derecha también lo es. Esto significa que se puede lograr que el if sea determinístico (solo una guarda verdadera).

$\{P\}$				$\{P\}$			
if	B_0	$\to$	S_0	**if**	$B_0 \wedge \neg B_1$	$\to$	S_0
[]	B_1	$\to$	S_1	[]	B_1	$\to$	S_1
fi				**fi**			
$\{Q\}$				$\{Q\}$			

14.12 Demostrar que todo ciclo con más de una guarda puede reescribirse como un ciclo con una sola guarda:

Probar que el bucle

do	B_0	$\to S_0$
[]	B_1	$\to S_1$
od		

puede reescribirse como

do	$B_0 \vee B_1$	$\to$	**if** $B_0 \to S_0$
			[] $B_1 \to S_1$
			fi
od			

Este resultado es útil, pues no todos los lenguajes de programación admiten ciclos con varias guardas.

14.13 Tomando como invariante $P : x \leq n \wedge y = 2^x$ y como función cota $t : n - x$ probar que el siguiente programa es correcto:

|[**var** x, y, n : **Integer**
$\{n \geq 0\}$
$x, y := 0, 1$
do $x \neq n \to x, y := x + 1, y + y$ **od**
$\{y = 2^n\}$
]|

14.14 Probar que el siguiente programa es correcto tomando como invariante $P : x \geq 0 \wedge y \leq n$ y como función cota $t : x + 2 \times (n - y)$.

```
|[  var x, y, n : Integer
    {n ≥ 0}
    x, y := 0, 0
    do x ≠ 0 → x := x − 1
    []  y ≠ n → x, y := x + 1, y + 1
    od
    {x = 0 ∧ y = n}
]|
```

14.15 Probar la corrección del siguiente bucle, tomando $P : 1 \leq k \leq n \wedge b = fib.(k-1) \wedge c = fib.k$ y $t : n - k$.

```
|[  var k, b, c : Integer;
    {n > 0}
    k, b, c := 1, 0, 1
    do k ≠ n → k, b, c := k + 1, c, b + c od
    {c = fib.n}
]|
```

donde fib es la función de Fibonacci. Explicar el propósito del programa.

14.16 Probar la corrección del siguiente bucle, tomando $P : 0 \leq i \leq n+1 \wedge \neg(\exists\, j : 0 \leq j < i : x = b[j])$ y función cota $t : n - i$.

```
|[  const n : Integer;
    var i, x : Integer;
        b : array[0 . . . n] of Integer;
    {n ≥ 0}
    i := 0
    do i ≤ n ∧ x ≠ b[i] → i := i + 1 od
    {(0 ≤ i ≤ n ∧ x = b[i]) ∨ (i = n + 1 ∧ ¬(∃ j : 0 ≤ j ≤ n : x = b[j]))}
]|
```

Explicar el propósito del programa.