Capítulo 13
La Programación Imperativa

Índice del Capítulo

13.1. Introducción

A partir de ahora cambiaremos el modelo de computación subyacente, y por lo tanto el formalismo para expresar los programas. Esto no significa que lo hecho hasta ahora no nos será útil en lo que sigue, ya que la experiencia adquirida en el cálculo de programas, y la consecuente habilidad para el manejo de fórmulas, seguirá siendo escencial en este caso, mostrando así que pese a sus diferencias, la programación funcional y la imperativa tienen varias cosas en común.

13.2. Especificaciones de programas

Los problemas que se presentan a un programador están, en general, expresados de manera informal, lo cual resulta demasiado impreciso a la hora de programar. Por eso es necesario precisar los problemas por medio de especificaciones formales. En la primera parte del curso se especificaba, usando lógica y un cálculo de funciones, el valor que debía calcularse. Este tipo de especificaciones era adecuado para el desarrollo de programas funcionales. En el estilo de

programación imperativa, la computación no se expresa como cálculo de valores sino como modificación de estados. Para desarrollar programas imperativos es por lo tanto deseable expresar las especificaciones como relaciones entre estados iniciales y finales.

Así es que un programa en el modelo de la programación imperativa estará especificado mediante:

- Una precondición: expresión booleana que describe los estados iniciales de las variables del programa, para los cuales se define el mismo.
- Una postcondición: expresión booleana que describe los estados finales del programa luego de su ejecución.

La notación que utilizaremos para expresar la especificación de un programa imperativo es:

$$\{P\}\ S\ \{Q\}$$

donde P y Q son predicados y S es un programa (o secuencia de comandos), tiene la siguiente interpretación:

> Siempre que se ejecuta S comenzando en un estado que satisface P, se termina en un estado que satisface Q.

P es llamada precondición de S, y Q su postcondición. La terna $\{P\}\ S\ \{Q\}$ se denomina tripleta de Hoare.

Derivar un programa imperativo consiste en encontrar S que satisfaga las especificación dada, es decir, consiste en encontrar S tal que, aplicado a un estado que satisfaga P, termine en un estado que satisfaga Q. Mientras que una verificación de que S es correcto (respecto a su especificación) consiste en probar que la tripleta $\{P\}\ S\ \{Q\}$ es verdadera.

Cuando querramos dar solo la especificación de un programa, no utilizaremos la tripleta $\{P\}\ S\ \{Q\}$, dado que esta notación no indica qué variables cambiaron en S. Por ejemplo, $\{true\}\ S\ \{x = y\}$, dice que S termina en un estado que satisface $x = y$, pero no cuáles variables entre x e y cambiaron luego de la ejecución.

Denotaremos formalmente una especificación como:

$$\{P\}\ x :=?\ \{Q\} \tag{13.1}$$

donde P es la precondición del programa, Q la postcondición, y x es la lista de variables que pueden cambiar de valor.

En los lenguajes imperativos, también es necesario explicitar el tipo de las variables y constantes del programa, lo cual se hace a través de una *declaración de variables y constantes*. Esto completa la especificación del programa a la vez que indica implícitamente las operaciones que se pueden realizar con las variables y constantes declaradas.

Como ejemplo consideremos la siguiente especificación de un programa que calcula el máximo de las variables x e y.

$$
\begin{array}{l}
|[\quad \text{var } x, y, z : \mathit{Integer}; \\
\quad \{\mathit{true}\} \\
\quad z :=? \\
\quad \{z = \mathit{max}(x, y)\} \\
]|
\end{array}
$$

13.2.1. Representación de variables iniciales y finales

El siguiente programa:

$$t := x; x := y; y := t$$

intercambia los valores de las variables x e y, usando una variable local t. Para poder especificar este programa, necesitamos una forma de poder describir los valores iniciales de las variables x e y. Usaremos para ello dos *variables de especificación* que llamamos X e Y. La especificación del programa, junto con el programa mismo sería:

$$\{x = X \wedge y = Y\}\ \ t := x; x := y; y := t\ \ \{x = Y \wedge y = X\}$$

Es importante notar, que estas variables no son variables del programa , por lo tanto no pueden aparecer en sentencias del mismo. Sólo pueden ocurrir en los predicados que forman la pre y post condición. En estos predicados pueden aparecer tres tipos de nombres: las variables de especificación, que denotaremos con letras mayúsculas; las variables libres del programa y variables de cuantificación.

A continuación se verán algunos ejemplos de especificación de programas, usando la notación 13.1. En el capítulo siguiente obtendremos los programas que satisfacen las especificaciones dadas, por ahora sólo nos abocaremos a definir las pre y post condiciones.

(13.1) Ejemplo. Especificaremos un programa que dadas dos variables enteras no negativas a y b, calcula el producto de ambas, mediante los siguientes predicados:

$$\left|[\quad \text{var } a, b, z : \mathit{Integer};\quad \{a \geq 0 \wedge b \geq 0\}\ \ z :=?\ \ \{z = a \times b\}\quad]\right|$$

El producto de a y b tiene que ser almacenado en alguna variable, elegimos z.

(13.2) Ejemplo. Especificaremos ahora un programa que dada una variable entera n, calcula la suma de los n primeros números naturales y almacena el resultado en n.

$$
\begin{array}{ll}
|[& \text{var } n : \mathit{Integer}; \\
& \{n \geq 0 \wedge n = N\} \\
& n :=? \\
& \{n = (\sum\ i : 0 < i < N : i)\} \\
]| &
\end{array}
$$

(13.3) Ejemplo.

Los arreglos (arrays en inglés) son estructuras de datos que proveen la mayoría de los lenguajes de programación imperativos. Un arreglo puede pensarse como una lista de tamaño fijo. Tradicionalmente, un arreglo es considerado un conjunto de variables independientes del mismo tipo, las cuales comparten un mismo nombre.

Utilizaremos la siguiente notación para describir un arreglo de $n+1$ elementos: $b[0, \ldots, n]$. El acceso a los elementos es similar al de listas, se realiza mediante índices. Por ejemplo, $b[0]$ corresponde al primer elemento del arreglo b, y para un arreglo de $n + 1$ elementos, $b[n]$, corresponde al último elemento.

Especificaremos un programa que dado un entero no negativo n y un arreglo $b[0, \ldots, n]$, calcula el producto de los primeros $n + 1$ elementos del arreglo.

$$
\begin{array}{ll}
|[& \text{const } n : \mathit{Integer}; \\
& \text{var } z : \mathit{Integer}; \\
& \text{var } b : \text{array } [0 \ldots n] \text{ of } \mathit{Integer}; \\
& \{n \geq 0\} \\
& z :=? \\
& \{z = (\prod\ i : 0 \leq i \leq n : b[i])\} \\
]| &
\end{array}
$$

(13.4) Ejemplo. Por último daremos la especificación de un programa que dados una variable entera n y un arreglo $b[0, \ldots, n]$ de enteros, ordena el arreglo de forma creciente.

$$
\begin{array}{ll}
|[& \text{const } n : \mathit{Integer}; \\
& \text{var } b : \text{array } [0 \ldots n] \text{ of } \mathit{Integer}; \\
& \{n \geq 0 \wedge b = B\} \\
& b :=? \\
& \{perm(b, B)\ \wedge\ (\forall\, i : 0 \leq i < n : b[i] \leq b[i + 1])\} \\
]| &
\end{array}
$$

donde *perm* es un predicado que toma dos arreglos y determina si uno es una permutación del otro. Dejamos como ejercicio la definición de este predicado.

Antes de continuar con el desarrollo de programas, veremos algunas reglas para la correcta interpretación y utilización de tripletas de Hoare.

13.3. Leyes sobre tripletas

El primer teorema, conocido como *ley de exclusión de milagros*, expresa:

(13.1) Teorema.

$$\{P\}\ S\ \{false\} \equiv P \equiv false$$

Esta ley establece que para cualquier programa S, si se requiere que termine y que los estados finales satisfagan $false$, entonces ese programa no puede ejecutarse exitosamente para ningún estado inicial. O bien, leído de otra manera, no existe ningún estado tal que si se ejecuta un programa S comenzando en él se puede terminar en un estado que satisfaga $false$.

La expresión

$$\{P\}\ S\ \{true\}$$

indica que, cada vez que comienza en un estado que satisface P, la ejecución de S concluye en un estado que satisface el predicado *true*. Esto se interpreta diciendo que la ejecución de S *termina* cada vez que se comienza en un estado que satisface P.

Las leyes que siguen a continuación expresan el hecho de que una precondición puede ser fortalecida y una postcondición puede ser debilitada, lo cual formulamos de la siguiente manera:

(13.2) Teorema.

$$\{P\}\ S\ \{Q\} \wedge (P_o \Rightarrow P) \Rightarrow \{P_o\}\ S\ \{Q\}$$

(13.3) Teorema.

$$\{P\}\ S\ \{Q\} \wedge (Q \Rightarrow Q_o) \Rightarrow \{P\}\ S\ \{Q_o\}$$

Se dice que una expresión P_0 es más fuerte que P si y solo si $P_0 \Rightarrow P$. Recíprocamente, P_0 es más débil que P si y solo si $P \Rightarrow P_0$. Con lo cual, la primera regla dice que si P_0 es más fuerte que P y vale $\{P\}\ S\ \{Q\}$, entonces $\{P_0\}\ S\ \{Q\}$ también es válida.

La segunda regla dice que si Q_0 es más débil que Q y vale $\{P\}\ S\ \{Q\}$, entonces también vale $\{P\}\ S\ \{Q_0\}$.

Supongamos que las tripletas $\{P\}\ S\ \{Q\}$ y $\{P\}\ S\ \{R\}$ son válidas. Entonces la ejecución de S en un estado que satisface P termina en uno que satisface Q y R, por lo tanto termina en un estado que satisface $Q \wedge R$. Esta ley se expresa de la siguiente manera:

(13.4) Teorema.

$$\{P\}\ S\ \{Q\} \wedge \{P\}\ S\ \{R\} \equiv \{P\}\ S\ \{Q \wedge R\}$$

Por último, si las ternas $\{P\}\ S\ \{Q\}$ y $\{R\}\ S\ \{Q\}$ son válidas, entonces al ejecutar S comenzando en un estado que satisface o bien P o bien R, es decir, $P \vee R$ termina en un estado que satisface Q; y por otro lado, si cada vez que se comienza en un estado que satisface $P \vee R$ al terminar vale Q entonces a fortiori (fortalecimiento de la condición) vale que comenzando en un estado que satisface P (o R) al terminar valdrá Q.

(13.5) Teorema.

$$\{P\}\ S\ \{Q\} \land \{R\}\ S\ \{Q\} \equiv \{P \lor R\}\ S\ \{Q\}$$

13.4. El transformador de predicados wp

Para cada comando S se puede definir una función $wp.S : Predicados \to Predicados$ tal que si Q es un predicado, $wp.S.Q$ representa el predicado más débil para el cual vale $\{P\}\ S\ \{Q\}$.

Para cada Q, $wp.S.Q$ se denomina precondición más débil de S con respecto a Q. En otras palabra, para todo predicado Q, $wp.S.Q = P$ si y solo si $\{P\}\ S\ \{Q\}$ y para cualquier predicado P_0 tal que $\{P_0\}\ S\ \{Q\}$ vale $P_0 \Rightarrow P$ (P es más débil que P_0).

$$(\forall Q : Predicados :: wp.S.Q = P \equiv \{P\}\ S\ \{Q\} \land (\forall P_0 : Predicados :: P_0 \Rightarrow P))$$

La definición de $wp.S$ permite relacionar las expresiones $\{P\}\ S\ \{Q\}$ y $wp.S.Q$ mediante el siguiente teorema:

(13.6) Teorema.

$$\{P\}\ S\ \{Q\}\ \equiv\ P \Rightarrow wp.S.Q$$

De esta forma podemos decir que el programa, $\{P\}\,S\,\{Q\}$, es correcto respecto a su especificación si $P \Rightarrow wp.S.Q$ es una expresión válida, es decir, es un teorema. Esto nos permitirá, una vez definido wp para todos los comandos posibles, trabajar en el conocido *cálculo de predicados*.

Observemos que $\{wp.S.Q\}S\{Q\}$ es una Terna de Hoare válida,

$$\begin{array}{ll} & \{wp.S.Q\}S\{Q\} \\ = & \quad \langle \text{ Teo. Hoare-wp 13.6 } \rangle \\ & wp.S.Q \Rightarrow wp.S.Q \\ = & \quad \langle \text{ Reflexividad de } \Rightarrow \rangle \\ & true \end{array}$$

y por lo tanto, siempre podremos utilizar $wp.S.Q$ como una precondición válida para un programa S con una precondición Q.

13.4.1. Propiedades de la *wp*

Las siguientes tres reglas, nos describen el comportamiento de este operador cuando el predicado sobre el cual lo aplicamos adopta el valor $false$, o las estructuras correspondiente a una conjunción o una disyución. En estos últimos casos,la distributividad del operador wp resulta válida mediante una equivalencia cuando el predicado es una conjunción, mientras que para disyución, sólo resulta válida la implicancia.

wp-Exc.Milagros $wp.S.false \equiv false$

wp-Distributividad con la Conjunción $wp.S.Q \wedge wp.S.R \equiv wp.S.(Q \wedge R)$

wp-Distributividad con la Disyunción $wp.S.Q \vee wp.S.R \Rightarrow wp.S.(Q \vee R)$

Existe una regla mas, la cual describe el comportamiento de la wp ante la operación $\Rightarrow$, para el cual sabemos que es monotona.

wp-Monotonía $(P \Rightarrow Q) \Rightarrow wp.S.P \Rightarrow wp.S.Q$

La prueba de esta propiedad se desprende de las anteriores reglas. Mostraremos a continuación como procedemos, emplearemos para ello la técnica de suposición del antecedente:

$$(P \Rightarrow Q) \Rightarrow wp.S.P \Rightarrow wp.S.Q$$

Supongo $P \Rightarrow Q \quad (S_1)$
Demuestro $wp.S.P \Rightarrow wp.S.Q$

$wp.S.P \Rightarrow wp.S.Q$
= $\langle\ S_1$; Def. alt $\Rightarrow$: $p \Rightarrow q \equiv p \wedge q \equiv p\ \rangle$
$wp.S.(P \wedge Q) \Rightarrow wp.S.Q$
= $\langle$ Distributividad wp sobre $\wedge\ \rangle$
$wp.S.P \wedge wp.S.Q \Rightarrow wp.S.Q$–teo. Deb. Fort 3.75b) con $p, q := wp.S.Q, wp.S.P$

Las reglas que presentamos en la sección anterior pueden demostrarse a partir de las propiedades de la $wp.S.Q$, veamos por ejemplo, como demostrar la regla 13.1. Es decir,

$$\{P\}\ S\ \{false\} \equiv P \equiv false$$

Partimos desde el lado izquierdo,

$\{P\}\ S\ \{false\}$
= $\langle$ Teo Hoare-wp 13.6 $\rangle$
$P \Rightarrow wp.S.false$
= $\langle$ Regla de wp-Exc. de Milagros $\rangle$
$P \Rightarrow false$
= $\langle$ Reducción al absurdo $\rangle$
$\neg P$
= $\langle$ Teo. 3.15 $\neg p \equiv p \equiv false\ \rangle$
$P \equiv false$ es el lado derecho.

Las restantes demostraciones se dejan como ejercicio.

13.5. Ejercicios

13.1 Escribir especificaciones para los programas que realizan las siguientes tareas:

1. Establece que la variable entera x sera igual al valor 7.
2. Almacena en z el valor absoluto de la variable entera x.
3. Almacena en la variable entera z su propio valor absoluto.
4. Almacena en las variables enteras q y r el resto y el cociente de la división entera entre las variables x por y, siendo x no negativa e y positiva.
5. Almacena en la variable x la raíz cuadrada entera de la constante n.
6. Dada tres variables reales , copia el valor de la suma de las tres en todas ellas.
7. Dado un entero n, con $n > 0$, el programa almacena en la variable x el mayor entero múltiplo de tres menor que n.
8. Dadas dos variables enteras, copia el valor de la primera en la segunda.
9. Dadas tres variables enteras, devuelve el máximo de las tres.
10. Dada una variable entera x, cuyo valor es mayor a 1, determina si x almacena un número primo.

13.2 Especificar los programas que realizan las actividades detalladas a continuación. En los mismos, se requerirá declarar variables del tipo arreglo (**array**$[0\,..\,n]$ **of** T). Recuerde que, en el caso que el problema no indique la longitud del arreglo, se deberá definir una constante de tipo **Integer** para darle una longitud especifica al mismo.

1. Dados un entero n y x, y un arreglo b a valores enteros, el programa almacena en la variable p si encontró el valor x en b.
2. Dada una variable entera $n \geq 0$ y un arreglo $b[0, \ldots, n]$, almancena en cada posición del arreglo la suma de todos los elementos del mismo.
3. Dado un arreglo entero b el programa almacena en la variable x al máximo valor de b.
4. Dado un arreglo b de tres elementos el programa almacena en todas las posiciones el valor 0.
5. Dado un arreglo real b de $n + 1$ elementos el programa actualiza cada posición del arreglo con el doble del producto de todos sus elementos.
6. Dado un arreglo de caracteres b de $n + 1$ elementos el programa modifica cada posición impar del arreglo asignándole en la misma la letra $'a'$.
7. Dado un arreglo de elementos booleanos, el programa altera cada elementos del mismo asignándole el valor opuesto.

8. Dado un arreglo de caracteres b de longitud 13, el programa almacena en la variable entera z la cantidad de vocales que aparecen en el arreglo.

13.3 Dar el significado operativo de las tripletas: $\{true\}\ S\ \{true\}$ y $\{false\}\ S\ \{true\}$

13.4 Deducir a partir de las leyes dadas en la sección 13.3 que

$$\{P_0\}\ S\ \{Q_0\}\ \text{ y }\ \{P_1\}\ S\ \{Q_1\}$$

implican

$$\{P_0 \wedge P_1\}\ S\ \{Q_0 \wedge Q_1\}\ \text{ y }\ \{P_0 \vee P_1\}\ S\ \{Q_0 \vee Q_1\}$$

13.5 Demostrar que $\{false\}\ S\ \{P\}$ es válida para cualquier S y cualquier P.

13.6 Demostrar que las leyes vistas en la sección 13.3, pueden demostrarse a partir de leyes para $wp.S$ enunciadas en la sección 13.4.1.