2

Verificación

2.1. Introducción

En el capítulo anterior hemos visto cómo utilizar predicados para definir conjuntos de estados y hemos especificado algoritmos mediante dos predicados llamados precondición y postcondición. El primero describe el conjunto de estados válidos al comienzo del algoritmo y el segundo el conjunto de estados alcanzables a su terminación. Ahora vamos a aplicar la misma idea a la descripción del conjunto de estados alcanzados por el algoritmo en un punto intermedio de su ejecución.

Escribiendo una aserción entre cada dos instrucciones elementales del algoritmo obtenemos un modo de razonar sobre la corrección de un programa. Comenzaremos considerando un algoritmo A de la forma A_1 ; ... ; A_n (siendo cada A_i una instrucción elemental) cuya especificación es $\{P\}\, A\, \{Q\}$. Al introducir predicados intermedios, obtenemos

$$\{P \equiv R_0\}\, A_1 \{R_1\}\ ;\ \ldots\ ;\ \{R_{n-1}\}\, A_n\, \{R_n \equiv Q\}.$$

Si el predicado inicial P se satisface y cada "algoritmo" elemental A_i consistente en una sola instrucción satisface la especificación $\{R_{i-1}\}\, A_i\, \{R_i\}$, entonces se satisface finalmente la postcondición Q y el algoritmo completo es correcto (cumple su especificación).

Necesitamos, por tanto, reglas para decidir si una instrucción elemental A satisface una especificación dada $\{P\}\, A\, \{Q\}$. En este capítulo vamos a ver dichas reglas para cada una de las instrucciones de un pseudocódigo imperativo con el que escribiremos nuestros algoritmos.

Conocer dichas reglas para cada instrucción del lenguaje es equivalente a conocer la *semántica* de este; de hecho, el conjunto de tales reglas se denomina *semántica axiomática* del lenguaje. Dichas reglas formalizan el efecto que, intuitivamente, sabemos tienen las instrucciones del lenguaje. A

estas reglas las llamaremos *reglas de verificación* porque nos permiten *verificar* la corrección de un algoritmo con respecto a su especificación. Tendrán la siguiente forma:

$$\frac{\textit{Premisas}}{\textit{Conclusión}}$$

Antes de ver las reglas para cada una de las instrucciones, proporcionamos unas reglas de verificación básicas.

2.1.1. Reglas básicas de verificación

Estas reglas son generales, es decir, se pueden aplicar independientemente del algoritmo A que aparece en ellas.

Fortalecimiento de la precondición. El hecho de que una especificación sea satisfecha por un algoritmo nos aporta información sobre otras especificaciones que el mismo algoritmo puede cumplir, por ejemplo aquellas en las que se fortalece la precondición:

$$\frac{P' \Rightarrow P \qquad \{P\}\, A\, \{Q\}}{\{P'\}\, A\, \{Q\}}$$

Debilitamiento de la postcondición. También podemos debilitar la postcondición y la especificación seguirá siendo correcta:

$$\frac{\{P\}\, A\, \{Q\} \qquad Q \Rightarrow Q'}{\{P\}\, A\, \{Q'\}}$$

Conjunción en la postcondición. La siguiente regla es bastante intuitiva:

$$\frac{\{P\}\, A\, \{Q_1\} \qquad \{P\}\, A\, \{Q_2\}}{\{P\}\, A\, \{Q_1 \wedge Q_2\}}$$

Si comenzando la ejecución en un estado que satisface P alcanzamos un estado que satisface Q_1 y también Q_2, entonces dicho estado satisface trivialmente su conjunción.

Obsérvese que no es necesario proporcionar una regla para la conjunción en la precondición puesto que es un caso particular de la regla de fortalecimiento de la precondición.

Disyunción en la precondición. La siguiente regla también es intuitiva:

$$\frac{\{P_1\}\, A\, \{Q\} \qquad \{P_2\}\, A\, \{Q\}}{\{P_1 \vee P_2\}\, A\, \{Q\}}$$

Si podemos alcanzar un estado que satisface Q cuando partimos de un estado que satisface P_1 o de uno que satisface P_2, entonces también podremos alcanzarlo a partir de uno que satisface la disyunción.

Obsérvese que no es necesario proporcionar una regla para la disyunción en la postcondición puesto que se trata de un caso particular de la regla de debilitamiento de la postcondición.

2.1.2. Precondición más débil

Acabamos de ver que la precondición de una especificación correcta se puede fortalecer. Esto implica que para demostrar la corrección de una especificación $\{P\}\ A\ \{Q\}$ basta con determinar cuál es la precondición más débil R que cumple $\{R\}\ A\ \{Q\}$ y comprobar que P es más fuerte que R. Si denotamos a la *precondición más débil* de A con respecto a Q como $\mathsf{pmd}(A, Q)$ podemos aplicar siempre la siguiente regla:

$$\frac{P \Rightarrow \mathsf{pmd}(A, Q)}{\{P\}\ A\ \{Q\}}$$

A continuación proporcionamos para cada una de las instrucciones del lenguaje en que escribiremos nuestros algoritmos

- la regla de verificación para dicha instrucción y
- la forma de determinar la precondición más débil de la instrucción con respecto a una postcondición dada.

2.1.3. Reglas específicas

Instrucción nula. La instrucción **nada** no hace nada, como su nombre indica. El axioma que define su comportamiento es el siguiente:

$$\{P\}\ \textbf{nada}\ \{P\}.$$

Los conjuntos de estados inicial y final son el mismo, lo que nos indica que la instrucción siempre termina y que su efecto sobre el estado del cómputo es nulo. Por tanto, la precondición más débil para la instrucción **nada** se obtiene de la siguiente manera:

$$\mathsf{pmd}(\textbf{nada}, Q) \iff Q.$$

Asignación simple. La instrucción de asignación

$$x := e$$

donde x es una variable y e una expresión del mismo tipo que x, tiene el siguiente significado intuitivo: la expresión e es evaluada en el estado actual y su valor es asignado a la variable x, que pierde su antiguo valor. El axioma para la asignación es el siguiente:

$$\{\,\mathsf{def}(e) \wedge Q_x^e\,\}\ x := e\ \{Q\},$$

donde $\mathsf{def}(e)$ denota el predicado que determina que el valor de la expresión e está bien definido. Por ejemplo, si e es la expresión aritmética x div y donde x e y son variables enteras, el predicado $\mathsf{def}(e)$ es $y \neq 0$ puesto que la división no está bien definida cuando el divisor es 0. Otras expresiones que típicamente pueden provocar errores en la evaluación de una expresión son aquellas en las que se accede a una posición de un vector, la cual debe estar dentro del rango del vector para que el acceso esté bien definido. En aquellos casos en

los que sea evidente que el predicado $\mathsf{def}(e)$ es equivalente a cierto omitiremos comentarios al respecto.

Si queremos que x satisfaga cierta propiedad Q después de ejecutarse la instrucción, dicha propiedad ha de ser satisfecha por e antes de ejecutarse aquella, dado que el valor de la expresión e antes de ejecutarse la instrucción va a ser copiado en x. Por tanto,

$$\mathsf{pmd}(x := e, Q) \iff \mathsf{def}(e) \wedge Q^e_x.$$

Asignación múltiple. La asignación múltiple modifica el valor de varias variables *simultáneamente*. La sintaxis general es

$$\langle x_1, \ldots, x_n \rangle := \langle e_1, \ldots, e_n \rangle$$

donde cada e_i es una expresión del tipo de la variable x_i. Al ejecutarse esta instrucción se evalúan *todas* las expresiones en el estado actual, sin modificar *ninguna* variable hasta que todas las expresiones estén evaluadas, momento en el que se asignan los correspondientes valores a cada variable.

El axioma para este tipo de asignación es

$$\left\{ \bigwedge_{i=1}^{n} \mathsf{def}(e_i) \wedge Q^{e_1,\ldots,e_n}_{x_1,\ldots,x_n} \right\} \langle x_1, \ldots, x_n \rangle := \langle e_1, \ldots, e_n \rangle \{Q\}.$$

Y la precondición más débil

$$\mathsf{pmd}(\langle x_1, \ldots, x_n \rangle := \langle e_1, \ldots, e_n \rangle, Q) \iff \bigwedge_{i=1}^{n} \mathsf{def}(e_i) \wedge Q^{e_1,\ldots,e_n}_{x_1,\ldots,x_n}.$$

Asignación a vectores. La regla de la asignación no se puede aplicar a asignaciones de la forma

$$v[i] := e$$

tratando a la expresión $v[i]$ como si fuese una variable. Si admitiésemos aplicar la regla de la asignación de este modo, tendríamos:

a) Enunciados verdaderos, pero imposibles de verificar (utilizando la regla de la asignación), como

$$\{i = j\}\ v[i] := 0\ \{v[j] = 0\}.$$

b) Enunciados falsos, pero que se pueden "verificar" (incorrectamente), como

$$\{v[1] = 2 \wedge v[2] = 2\}\ v[v[2]] := 1\ \{v[v[2]] = 1\}.$$

La regla correcta de la asignación a vectores se obtiene interpretando $v[i] := e$ como una asignación que modifica al vector v, es decir, viendo la asignación como

$$v := asig(v, i, e)$$

siendo el vector $asig(v, i, e)$ un vector del mismo tipo que v que cumple:

$$asig(v,i,e)[j] = \begin{cases} e & \text{si } i = j \\ v[j] & \text{si } i \neq j \text{ y } j \text{ está en el rango de los índices de } v. \end{cases}$$

Para razonar acerca de la asignación $v[i] := e$ aplicamos a $v := asig(v, i, e)$ la regla de la asignación normal.

Composición secuencial. La composición secuencial de dos instrucciones A_1 y A_2, denotada

$$A_1 \; ; \; A_2$$

se utiliza para encadenar (o "secuencializar") cómputos. Como consecuencia, el estado final producido por A_1 se convierte en el estado inicial de A_2. La siguiente regla expresa formalmente esta idea:

$$\frac{\{P\}\, A_1\, \{R\} \qquad \{R\}\, A_2\, \{Q\}}{\{P\}\, A_1 \; ; \; A_2\, \{Q\}}$$

Aparentemente es necesario inventar un predicado R que describa el conjunto de estados intermedios, alcanzados después de ejecutar A_1 y antes de ejecutar A_2, y demostrar que se satisfacen las especificaciones de cada instrucción por separado. En la mayoría de los casos vamos a tomar como R la $\mathsf{pmd}(A_2, Q)$, para solo demostrar $\{P\}\, A_1\, \{R\}$ calculando a su vez la $\mathsf{pmd}(A_1, R)$ y viendo que $P \Rightarrow \mathsf{pmd}(A_1, R)$. Por tanto,

$$\mathsf{pmd}(A_1 \; ; \; A_2, Q) \iff \mathsf{pmd}(A_1, \mathsf{pmd}(A_2, Q)).$$

Cuando tengamos más de dos instrucciones utilizaremos la misma idea, determinando aserciones intermedias y construyendo demostraciones para cada paso.

Composición alternativa. La sintaxis de la composición alternativa, o distinción de casos, es:

casos
$\quad B_1 \;\to\; A_1$
$\quad [] \; B_2 \;\to\; A_2$
$\quad \vdots$
$\quad [] \; B_n \;\to\; A_n$
fcasos

donde $A_1, \ldots, A_n$ son instrucciones, posiblemente composiciones secuenciales, y $B_1, \ldots, B_n$ son expresiones booleanas.

Se exige que se cumpla alguna de las guardas B_i y que estas sean excluyentes además de estar

bien definidas:

$$\frac{\begin{array}{c} P \;\Rightarrow\; \bigwedge_{i=1}^{n} \mathsf{def}(B_i) \\ P \;\Rightarrow\; \bigoplus_{i=1}^{n} B_i \\ \{P \wedge B_1\}\, A_1\, \{Q\} \\ \vdots \\ \{P \wedge B_n\}\, A_n\, \{Q\} \end{array}}{\{P\}\ \textbf{casos}\ B_1 \;\rightarrow\; A_1 \;\square\; \cdots \;\square\; B_n \;\rightarrow\; A_n\ \textbf{fcasos}\ \{Q\}}$$

El operador $\oplus$ denota la disyunción exclusiva.

La precondición más débil para la composición alternativa se calcula de la siguiente manera:

$$\begin{array}{l} \mathsf{pmd}(\textbf{casos}\ B_1 \;\rightarrow\; A_1 \;\square\; \cdots \;\square\; B_n \;\rightarrow\; A_n\ \textbf{fcasos}, Q) \\ \iff\; \bigwedge_{i=1}^{n} \mathsf{def}(B_i) \wedge \bigoplus_{i=1}^{n} B_i \wedge (B_1 \rightarrow \mathsf{pmd}(A_1, Q)) \wedge \ldots \wedge (B_n \rightarrow \mathsf{pmd}(A_n, Q)). \end{array}$$

Una forma alternativa de verificar la distinción de casos consiste en demostrar

$$P \wedge B_i \;\Rightarrow\; \mathsf{pmd}(A_i, Q)$$

para cada $i \in \{1..n\}$, además de comprobar que $P \;\Rightarrow\; \bigwedge_{i=1}^{n} \mathsf{def}(B_i) \wedge \bigoplus_{i=1}^{n} B_i$.

Un caso particular de la distinción de casos se presenta cuando tenemos

casos
  $B \;\rightarrow\; A_1$
$\square$ $\neg B \;\rightarrow\; A_2$
fcasos

que escribimos como

si B **entonces**
  A_1
si no
  A_2
fsi

La regla de verificación para este caso es

$$\frac{\begin{array}{c} P \;\Rightarrow\; \mathsf{def}(B) \\ \{P \wedge B\}\, A_1\, \{Q\} \\ \{P \wedge \neg B\}\, A_2\, \{Q\} \end{array}}{\{P\}\ \textbf{si}\ B\ \textbf{entonces}\ A_1\ \textbf{si no}\ A_2\ \textbf{fsi}\ \{Q\}}$$

y la precondición más débil

$$\begin{aligned} & \mathsf{pmd}(\textbf{si } B \textbf{ entonces } A_1 \textbf{ si no } A_2 \textbf{ fsi}, Q) \\ \iff\ & \mathsf{def}(B) \wedge ((B \wedge \mathsf{pmd}(A_1, Q)) \vee (\neg B \wedge \mathsf{pmd}(A_2, Q))). \end{aligned}$$

Obsérvese que este último predicado es equivalente a $(B \rightarrow \mathsf{pmd}(A_1, Q)) \wedge (\neg B \rightarrow \mathsf{pmd}(A_2, Q))$, obtenido calculando la precondición más débil de la distinción de casos equivalente.

Composición iterativa. La instrucción de iteración de nuestro lenguaje es la instrucción **mientras** cuya sintaxis es:

mientras B **hacer**
  A
fmientras

donde B es una expresión booleana y A una instrucción. La instrucción comienza evaluando la *condición de iteración* B: si es falsa, se finaliza la ejecución del bucle; en otro caso, se ejecuta el *cuerpo* A de la iteración y se vuelve a evaluar la condición B, repitiendo el proceso.

Para la verificación de un bucle necesitamos un predicado llamado *invariante* que describe los distintos estados por los que pasa el bucle, estableciendo la relación existente entre las variables que intervienen en este. El invariante lo escribiremos como anotación junto con el bucle:

$\{P\}$
$\{I\}$
mientras B **hacer**
  A
fmientras
$\{Q\}$

El invariante I tiene que cumplir las siguientes condiciones:

(i.1) Se satisface antes de empezar el bucle, es decir, antes de la primera iteración. Esto lo expresamos requiriendo que

$$P \Rightarrow I.$$

(i.2) Se mantiene al ejecutar el cuerpo A del bucle. El cuerpo se ejecuta solamente si la condición se cumple, es decir, tendrá que cumplirse

$$\{I \wedge B\}\, A\, \{I\}.$$

(i.3) Se cumple al salir del bucle, cuando B se hace falsa, y esto nos tiene que llevar a la postcondición:

$$I \wedge \neg B \Rightarrow Q.$$

Además de estos tres puntos, tendremos que probar que la instrucción iterativa *termina*. Para ello tendremos que buscar una función C que dependa de las variables del bucle (generalmente solo de aquellas variables que aparecen en la condición de iteración B) y que tome valores enteros, de forma que se cumpla que:

(c.1) Es mayor o igual que cero cuando se cumple la condición B:

$$I \wedge B \Rightarrow C \geq 0.$$

(c.2) Decrece al ejecutar el cuerpo A del bucle:

$$\{I \wedge B \wedge C = T\}\, A\, \{C < T\}.$$

Esta función es una cota superior del número de iteraciones que quedan por realizar y por eso nos referiremos a ella como *función de cota*.

De todo lo anterior concluimos la regla de verificación para la instrucción iterativa:

$$\frac{\begin{array}{c} P \Rightarrow I \\ \{I \wedge B\}\, A\, \{I\} \\ I \wedge \neg B \Rightarrow Q \\ I \wedge B \Rightarrow C \geq 0 \\ \{I \wedge B \wedge C = T\}\, A\, \{C < T\} \end{array}}{\{P\}\ \textbf{mientras}\ B\ \textbf{hacer}\ A\ \textbf{fmientras}\ \{Q\}}$$

2.1.4. Verificación de algoritmos recursivos

La recursividad es una característica de la mayoría de los lenguajes de programación que permite que un procedimiento o función haga referencia a sí mismo dentro de su definición. La recursividad se utiliza, al igual que la iteración, para describir cálculos que se repetirse cierto número de veces.

Razonar recursivamente permite resolver un problema P sobre unos datos D suponiendo que P ya está resuelto para otros datos D', del mismo tipo que D y en algún sentido más sencillos (caso recursivo). Para que el razonamiento pueda aplicarse se requiere que P pueda ser resuelto directamente para unos datos suficientemente simples (caso base).

Un algoritmo es *recursivo lineal* si su ejecución genera una única llamada recursiva. Un algoritmo es *recursivo múltiple* si una misma invocación genera más de una llamada recursiva.

El esquema general de un algoritmo recursivo lineal es

$\{\, P(\overline{x}) \,\}$
fun f-rec$(\overline{x} : T_1)$ **dev** $\langle\, \overline{y} : T_2 \,\rangle$
  casos
    $B_t(\overline{x}) \rightarrow \overline{y} := \mathsf{triv}(\overline{x})$
  □ $B_{nt}(\overline{x}) \rightarrow \overline{y} := \mathsf{c}(\text{f-rec}(\mathsf{s}(\overline{x})), \overline{x})$
  fcasos
ffun
$\{\, Q(\overline{x}, \overline{y}) \,\}$

donde: