Lógica Computacional

José Luis Fernández Vindel
Ángeles Manjarrés Riesco
Francisco Javier Díez Vegas

Dpto. Inteligencia Artificial
E.T.S.I. Informática
UNED
2003

Capítulo 4

VERIFICACIÓN DE PROGRAMAS SECUENCIALES

Resumen

Este tema estudia cómo definir con precisión la semántica de un lenguaje formal mediante la lógica de Hoare y cómo utilizar un método deductivo basado en ella para la verificación de programas secuenciales.

Objetivos

El objetivo principal es que el alumno aprenda a verificar pequeños programas secuenciales mediante la lógica de Hoare.

Metodología

Para poder comprobar que un programa es correcto hace falta establecer con precisión su semántica. Por ello, casi al principio de este tema se define un lenguaje que contiene sólo tres instrucciones: if-then-else, while-do *y* := *(asignación de valores a variables); aunque se trata de un lenguaje aparente muy simple, la mayor parte de las instrucciones de lenguajes de programación más complejos —salvo las relativas a interfaces de entrada/salida— se pueden construir a partir de estas tres instrucciones básicas. Luego se establece la semántica de este lenguaje mediante ciertas expresiones lógicas, las ternas de Hoare, que representan la transformación de estados asociada a cada instrucción. Finalmente, se expone un sistema deductivo para la lógica de Hoare y se explica cómo aplicarlo a la verificación y síntesis de programas.*

4.1 Introducción

El hecho de que un programa tenga un error puede resultar sumamente costoso, no sólo en términos económicos, sino que en ciertos casos incluso puede poner en peligro la vida de muchos seres humanos. Por eso es importante disponer de métodos que permitan comprobar que un programa cumple las especificaciones con que fue diseñado. Generalmente las especificaciones suelen venir dadas en lenguaje natural. Por ejemplo: “Quiero un programa que calcule las nóminas de mis empleados, a partir de los siguientes datos…”. Naturalmente, dada la ambigüedad y falta de precisión del lenguaje

natural y la ausencia de métodos que permitan una compresión automática del mismo, se hace necesario contar con descripciones formales que indiquen de forma precisa e inequívoca las especificaciones de cada programa. Una vez que se conocen las especificaciones, sería deseable contar con un método, implementable en un programa de ordenador, que generase automáticamente el programa buscado, libre de errores. Dado que esto es todavía hoy ciencia-ficción, al menos sería deseable contar con métodos que permitan comprobar de forma automática o semiautomática que cierto programa cumple las especificaciones, es decir, que hace lo que se espera de él, sin cometer nunca errores; es lo que se conoce como *verificación de programas*.

Aún estamos lejos de contar con verificadores totalmente automáticos, pero recientemente se han desarrollado ya verificadores semiautomáticos para lenguajes de alto nivel.[1] Los grandes avances que se han producido en las últimas tres décadas y los intereses de los fabricantes de software por garantizar la fiabilidad de sus productos hacen pensar que en los próximos años se seguirán produciendo progresos muy significativos en este campo. Precisamente por la importancia del tema, es de esperar que el conocimiento de los métodos de verificación formal sea una de las cualidades más valoradas de los ingenieros en informática de un futuro no muy lejano.

En este tema vamos a estudiar los métodos de verificación de programas secuenciales. En concreto, vamos a estudiar un sistema deductivo que permite verificar programas de un lenguaje de programación muy sencillo, tan sencillo que sólo tiene tres instrucciones: asignación de variables (`:=`), condicional (`if-then-else`), y bucle (`while-do`). Aunque se trata de un lenguaje aparentemente muy simple, la mayor parte de las instrucciones de lenguajes de programación más complejos —salvo las relativas a interfaces de entrada/salida— se pueden construir a partir de estas tres instrucciones básicas. Luego introduciremos ciertas expresiones lógicas, las ternas de Hoare, que permiten especificar formalmente un programa. Más adelante expondremos un sistema deductivo (el sistema de Hoare) y explicaremos cómo aplicarlo a la verificación de programas escritos en nuestro micro-lenguaje.

4.2 Sintaxis

4.2.1 Un micro-lenguaje de programación

Como hemos dicho en la introducción, la verificación de un programa escrito en un lenguaje de alto nivel es sumamente complicada. Para simplificar nuestro estudio, vamos a definir un pequeño lenguaje que sólo tiene tres instrucciones y dos tipos de datos: booleanos y enteros. En las conclusiones comentaremos las limitaciones de este micro-lenguaje en comparación con los lenguajes habituales.

Utilizando la notación gramatical de Naur Backus (NBF), definimos nuestro micro-lenguaje así:

$$E ::= n \mid x \mid \texttt{(}-E\texttt{)} \mid \texttt{(}E + E\texttt{)} \mid \texttt{(}E - E\texttt{)} \mid \texttt{(}E * E\texttt{)}$$

$$B ::= \texttt{true} \mid \texttt{false} \mid \texttt{!}B \mid \texttt{(}B \texttt{ \& } B\texttt{)} \mid \texttt{(}B \texttt{ | } B\texttt{)} \mid \texttt{(}E \texttt{ == } E\texttt{)} \mid \texttt{(}E \texttt{ != } E\texttt{)} \mid \texttt{(}E < E\texttt{)} \mid \texttt{(}E \texttt{ =< } E\texttt{)} \mid \texttt{(}E > E\texttt{)} \mid \texttt{(}E \texttt{ >= } E\texttt{)}$$

$$S ::= x\texttt{:=}E \mid \texttt{if } B \texttt{ then (}S\texttt{)[else (}S\texttt{)]} \mid \texttt{while } B \texttt{ do (}S\texttt{)} \mid S\texttt{;}S$$

Observe que en este lenguaje tenemos dos tipos de expresiones: boleanas (B) y enteras (E) y tres tipos de instrucciones (S, del inglés, *statement*). Una expresión booleana puede venir dada por un entero (n), tal como `1` ó `3758`, una variable de programa (por ejemplo, `x`), el opuesto de una expresión entera (por ejemplo, `-1` o `-y`) o bien por la suma, resta o multiplicación de dos expresiones enteras.

[1] Vea, por ejemplo, el proyecto KeY, `http://i12www.ira.uka.de/~key/`, que ha conseguido integrar *especificación formal* (en OCL) y *verificación semiautomática* (con una ampliación de las técnicas expuestas en este capítulo) en una herramienta CASE de *modelado mediante objetos* (UML), especialmente diseñada para programar en Java.

Las palabras reservadas `true` y `false` son expresiones boolenas. La negación de una expresión booleana también lo es, así como la conjunción y disyunción de expresiones booleanas y la comparación de expresiones enteras.[2]

El primer tipo de instrucción que tenemos es la asignación de una expresión numérica E a una variable x. Hay también expresiones condicionales (de la forma `if-then` o `if-then-else`) y, en tercer lugar, bucles (instrucciones `while`). La concatenación de dos expresiones (C;C) también es una expresión.

Ejemplo 4.1 El siguiente programa, que denominaremos `fact1`, está escrito en nuestro micro-lenguaje:

```
fact := 1;
i := 0;
while (i != x) do (
    i := i + 1;
    fact := fact * i
)
```

Como el lector habrá adivinado, este programa sirve para calcular el factorial de x. Más adelante vamos a demostrar formalmente que esto es así.

4.2.2 Especificación de estados

Dado el conjunto de variables que aparecen en un programa, un *estado* viene dado por la asignación de un valor a cada una de variables. Por ejemplo, si las variables son `x`, `i` y `fact`, la expresión $\{x = 10 \wedge i = 4 \wedge fact = 24\}$ representa un estado.

Puede haber también expresiones en que no aparezcan todas las variables que se usan en el programa. Tales expresiones no representan un estado, sino un conjunto infinito de ellos. Así, si las variables de un programa son `x`, `i` y `fact`, la expresión $\{i = 4\}$ no representa un único estado, sino el conjunto de todos los estados en que la variable `i` tiene el valor 4; este conjunto tiene tantos estados como valores puedan tomar `x` y `fact`. Otro ejemplo de expresión que representa un conjunto de estados es la siguiente: $\{x \geq 0 \wedge x \leq 10 \wedge i \neq x\}$.

Cualquier expresión lógica, por tanto, representa un estado o un conjunto de estados. En particular, $\{\top\}$ representa todos los estados posibles, y $\{\bot\}$ no representa ningún estado (conjunto vacío).

4.2.3 Ternas de Hoare

Dado un lenguaje que expresa los estados de un sistema y un lenguaje de programación (cada fórmula de este lenguaje representa una instrucción, es decir, un programa), definimos un nuevo lenguaje formado por ternas de la forma

$$\{\text{precondición}\}(\text{instrucción})\{\text{postcondición}\}$$

Estas expresiones se denominan ternas de Hoare. La interpretación intuitiva de esta expresión es que si un sistema que se encuentra en alguno de los estados representados por la precondición ejecuta la instrucción, pasa a alguno de los estados representados por la postcondición.

[2] Observe que en Pascal la asignación de variables se representa mediante "`:=`" y la comparación mediante "=", mientras que en C y Java la asignación se representa mediante "=" y la comparación mediante "==". Para evitar confusiones, en nuestro micro-lenguaje hemos utilizado "`:=`" para la asignación y "==" para la comparación. En cambio, en las expresiones lógicas utilizaremos el signo "=" para denotar la igualdad porque en ellas no hay confusión sobre su significado.

Ejemplo 4.2 La terna $\{i = 4\}(\texttt{i:=i+1})\{i = 5\}$ significa que si el sistema se encuentra en un estado en que `i` vale 4 y ejecuta la instrucción `i:=i+1`, pasa a un estado en que `i` vale 5.

Ejemplo 4.3 $\{i > 0\}(\texttt{i:=i+1})\{i > 1\}$.

Ejemplo 4.4 Dado que la condición $\top$ es cierta para todos los estados, la terna $\{\top\}(\texttt{i:=5})\{i = 5\}$ significa que, cualquiera que se sea el estado inicial del sistema, la ejecución de la instrucción `i:=5` hace que el sistema pase a un estado en que `i` vale 5.

Nota. Como hemos visto anteriormente al definir nuestro micro-lenguaje, la concatenación de dos instrucciones es una nueva instrucción. En la práctica llamamos *programa* a una instrucción compuesta que realiza cierta tarea. Sin embargo, formalmente no hay diferencia entre programas e instrucciones.

Cuando tenemos una terna de Hoare en que el programa no está definido todavía, se dice que tenemos una *especificación* del programa, pues estamos indicando solamente cuáles son los requisitos que debe cumplir el programa. Por ejemplo, la condición

$$\{x \geq 0\}(\texttt{S})\{fact = x!\} \tag{4.1}$$

es una especificación, porque indica que el programa debe calcular el factorial de un número no negativo (más adelante veremos por qué esta especificación no es del todo correcta).

4.2.4 Variables de programa y variables lógicas

Imagine que queremos expresar mediante una terna de Hoare la propiedad de que, cualquiera que sea el valor inicial de la variable `i`, la instrucción `i:=i+1` hace que el valor de esta variable aumente en una unidad. Por analogía con el ejemplo 4.4, un aprendiz de lógica ingenuo podría intentar representarla así:

$$\{\top\}(\texttt{i:=i+1})\{i = i + 1\} \qquad \text{(incorrecta)}$$

Claramente, esta expresión es incorrecta, porque la condición $i = i + 1$ siempre es falsa. La forma correcta de representar la propiedad anterior es ésta:

$$\forall a,\ \{i = a\}(\texttt{i:=i+1})\{i = a + 1\}$$

Observe que en este caso la variable a no aparece en la instrucción. Estas variables, que no forman parte del programa, sino que se introducen para relacionar la precondición con la postcondición, se denominan *variables lógicas*, para distinguirlas de las *variables de programa*, que son las que aparecen la instrucción S. En las ternas de Hoare, las variables lógicas están siempre sujetas a un cuantificador universal, que en la práctica suele omitirse, de modo que lo habitual será escribir la expresión anterior simplemente así:

$$\{i = a\}(\texttt{i:=i+1})\{i = a + 1\}$$

Pero no debemos olvidar que en realidad hay un cuantificador en esta expresión, aunque no lo hayamos escrito.

Veamos con otro ejemplo la necesidad de introducir variables lógicas. Ya hemos dicho antes que la expresión (4.1) es una especificación para el cálculo del factorial de enteros no negativos. Aunque todavía no hemos desarrollado los métodos formales, no será difícil para el lector comprobar que el

programa `fact1` del ejemplo 4.1 cumple esta condición, lo cual es correcto. Sin embargo, observe que el programa `(x:=1;fact:=1)` también cumple la condición (4.1),

$$\{x \geq 0\}(\texttt{x:=1;fact:=1})\{fact = x!\}$$

a pesar de que no calcula correctamente el factorial cuando $x > 1$. Esto nos muestra que la especificación (4.1) no es satisfactoria.

En cambio, el siguiente programa, que llamaremos `fact2`,

```
fact := 1;
while (x != 0) do (
    fact := fact * x;
    x = x - 1;
)
```

sí calcula correctamente el factorial, pero no cumple la condición (4.1). Por tanto, la especificación (4.1) no es necesaria ni suficiente para garantizar que el programa calcula el factorial.

¿Cuál es la forma correcta de representar esta especificación? Una solución es la siguiente:

$$\forall a,\ \{x = a\}(\texttt{S})\{fact = a!\} \tag{4.2}$$

En esta expresión hemos indicado explícitamente el cuantificador universal para que quede claro que a es una variable lógica, y por tanto no puede aparecer en el cuerpo del programa. (Si el programa utilizase la variable a tendríamos que escoger una variable lógica diferente, como es natural.) Recomedamos al lector que compruebe que el programa `(x:=1;fact:=1)` no cumple esta especificación, pero los programas `fact1` y `fact2` sí la cumplen.

La conclusión que se saca de los ejemplos anteriores es que hay que utilizar variables lógicas cuando queremos que la poscondición haga referencia al valor que toma cierta variable en la precondición y el cuerpo del programa modifica el valor de esa variable. Así, en nuestro ejemplo, queremos que, después de ejectutar el programa S, la variable *fact* contenga el factorial del valor asignado inicialmente a x. Cuando el programa no modifica el valor de x (como era el caso del programa `fact1`), la especificación (4.1) no plantea problemas, porque el valor final de x es el mismo que el inicial. En cambio, cuando el programa modifica el valor de x (como era el caso del programa `fact2`), la postcondición $\{fact = x!\}$, que hace referencia al valor **final** de x, no nos sirve, y por eso debemos utilizar la especificación (4.2).

4.3 Semántica de los programas

4.3.1 Corrección total

Una terna de Hoare, $\{p\}(S)\{q\}$, puede considerarse como una proposición y por tanto es posible asignarle un valor de verdad. Decimos que una terna de Hoare es cierta si todo sistema que parte de un estado (cualquiera) que satisface p pasa a un estado que satisface q, y se representa así:[3]

$$\vDash_{\text{tot}} \{p\}(S)\{q\}$$

[3] En este capítulo estamos definiendo la semántica de modo un tanto informal, basada en el estado inicial y el estado final de una computación. Para un tratamiento riguroso, el lector puede consultar el libro de Francez [1992] o el de Apt y Olderog [1997]. La definición de semántica que ofrece el libro de Ben-Ari [2001] es diferente, y está basada en el concepto de *condición más débil* ("weakest precondition").

Esta propiedad, denominada *corrección total*, es muy importante en la práctica, pues la verificación del programa S consiste precisamente en demostrar que siempre que el sistema satisface ciertas condiciones (p) la ejecución de S hace que se satisfaga la condición que nos interesa (q). Las dos tareas más frecuentes que vamos encontrar en la práctica son:

Verificación: Tenemos las condiciones p y el programa S (codificado por nosotros mismos o por otro programador) y queremos demostrar que el programa es correcto (es decir, la ejecución del programa hace que se satisfaga la condición q).

Programación: Conocemos las condiciones p y q y queremos encontrar el programa S. En este caso, se trata de un problema de programación.

En este capítulo vamos a estudiar métodos formales de verificación. En la sección 4.7.2 mencionaremos brevemente la posibilidad de utilizar estos métodos para realizar a la vez la programación y la verificación.

Para algunas instrucciones sencillas es fácil ver que se satisface la corrección total. Por ejemplo,

$$\vDash_{\text{tot}} \{i = 4\}(\texttt{i:=i+1})\{i = 5\}$$

En la práctica, demostrar la corrección total directamente suele ser bastante complicado. En estos casos podemos resolver el problema dividiéndolo en dos subtareas: (1) demostrar que el programa, **si termina**, llega a q —es lo que se denomina *corrección parcial*— y (2) demostrar que el programa termina.[4] En la práctica, el método que se sigue es demostrar primero la corrección parcial y adaptar luego la demostración para probar la corrección total (véase la sec. 4.6).

4.3.2 Corrección parcial

La *corrección parcial* se expresa así:

$$\vDash_{\text{par}} \{p\}(S)\{q\}$$

y, como hemos dicho, significa que si un sistema que se encuentra en un estado que satisface la condición ejecuta el programa S, **si el programa termina**, llega al estado q.

De esta definición se deduce que un programa que no termina nunca siempre es *parcialmente correcto* (para toda p y toda q), pero nunca es *totalmente correcto*. Por ejemplo,

$$\vDash_{\text{par}} \{p\}(\texttt{while true do (i:=0)})\{q\}$$
$$\nvDash_{\text{tot}} \{p\}(\texttt{while true do (i:=0)})\{q\}$$

Otro ejemplo trivial es que para todo S se cumple que

$$\vDash_{\text{par}} \{\bot\}(S)\{q\} \tag{4.3}$$

Por tanto, todo programa S es *parcialmente correcto* para la precondición "$\bot$" cualquiera que sea la postcondición.[5]

Análogamente tenemos que

$$\vDash_{\text{par}} \{p\}(S)\{\top\} \tag{4.4}$$

[4] Obviamente, las únicas instrucciones que pueden hacer que el programa no termine son los bucles `while`. Por eso, para demostrar que un programa termina basta demostrar que todos sus bucles terminan.

[5] Naturalmente, este ejemplo sólo tiene interés didáctico, porque la precondición "$\bot$" excluye todos los estados. Dado que un ordenador real siempre va a encontrarse en algún estado, la propiedad (4.3) nunca puede aplicarse en la práctica.

porque la condición "$\top$" se satisface siempre, y de ahí se deduce que todo programa es *parcialmente correcto* para la prostcondición "$\top$" cualquiera que sea la precondición.[6]

En la próxima sección vamos a estudiar un sistema deductivo que nos permitirá demostrar la corrección parcial de programas que presentan interés real. En la sección 4.6 discutiremos cómo demostrar la terminación en el caso de programas con bucles.

4.4 El sistema deductivo de Hoare

El sistema deductivo de Hoare se basa en cinco axiomas, que sirven como reglas de deducción:

Asignación:

$$\vdash \{P(E)\}(\texttt{x:=}E)\{P(x)\} \tag{4.5}$$

Condicional:

$$\frac{\vdash \{p \wedge B\}(S_1)\{q\} \qquad \vdash \{p \wedge \neg B\}(S_2)\{q\}}{\vdash \{p\}(\texttt{if } B \texttt{ then } (S_1) \texttt{ else } (S_2))\{q\}} \tag{4.6}$$

Bucle:

$$\frac{\vdash \{p \wedge B\}(S)\{p\}}{\vdash \{p\}(\texttt{while } B \texttt{ do } (S))\{p \wedge \neg B\}} \tag{4.7}$$

Composición:

$$\frac{\vdash \{p\}(S_1)\{q\} \qquad \vdash \{q\}(S_2)\{r\}}{\vdash \{p\}(S_1 ; S_2)\{r\}} \tag{4.8}$$

Encadenamiento:

$$\frac{\vdash \{p \to p'\} \qquad \vdash \{p'\}(S)\{q'\} \qquad \vdash \{q' \to q\}}{\vdash \{p\}(S)\{q\}} \tag{4.9}$$

A estas reglas habría que añadir los axiomas propios del dominio. En el caso de nuestro microlenguaje el dominio es la aritmética entera, pues el único tipo de datos que admite son enteros (a parte de expresiones booleanas, naturalmente, que formarn parte de la lógica). Por ejemplo, un axioma del dominio puede ser "$\forall x,\ x = x$". Otro axioma puede ser "$\forall x, \forall y, \forall z,\ x = y \to x + z = y + z$". De hecho, más adelante veremos que en la verificación de programas se combinan dos sistemas deductivos, uno para razonar sobre las ternas de Hoare y otro para razonar sobre el dominio (la aritmética).

Vamos a explicar a continuación cada una de estas reglas. Pero antes debemos mencionar que los nombres varían mucho de un texto a otro. Por eso no es importante recordar los nombres de las reglas sino su significado y la forma en que se aplican.

4.4.1 Regla de asignación

La regla de asignación nos dice que si una condición, expresada en forma de predicado, se cumple para la expresión E entonces se cumple también para x después de haber asignado a la variable `x` el valor E (`x:=E`). Dicho de otra forma, si queremos demostrar que la variable x cumple el predicado P tenemos que demostrar que el valor (dado por la expresión E) que hemos asignado a x cumplía el predicado P, porque si no se cumplía $P(E)$, tampoco va a cumplir $P(x)$.

[6] De nuevo encontramos un ejemplo que sólo tiene interés didáctico, a pesar de que la propiedad (4.4) siempre es cierta: la razón es que esta propiedad sólo nos dice que, si el programa S termina, el sistema se va a encontrar *en algún estado*, lo cual es tanto como no decir nada.

Ejemplo 4.5 La regla de asignación nos permite deducir que

$$\vdash \{a = 0\}(\texttt{x:=a})\{x = 0\}$$

En este ejemplo, el predicado P es "igual a cero" y la expresión E (el valor asignado a la variable) es "a"; por eso $P(E)$ es "$a = 0$". El significado de la regla de asignación, en este ejemplo, es que si a valía inicialmente 0, la instrucción `x:=a` hace que x también valga 0.

En la práctica esta regla se aplica hacia atrás, de modo que si queremos demostrar que después de la asignación `x:=E` se cumple $P(x)$, intentaremos demostrar que $P(E)$ era cierto antes de la asignación. La forma de hacer esto consiste en tomar la postcondición y sustituir en ella x por E. Así, en el ejemplo anterior, debemos tomar la postcondición "$x = 0$", y sustituir x por el valor que le asigna la instrucción (`x:=a`), con lo cual obtenemos la precondición "$a = 0$".

Ejemplo 4.6 Si queremos ver qué precondición satisface esta terna

$$\{?\}(\texttt{x:=a+1})\{x = 0\}$$

tenemos que tomar la postcondición, "$x = 0$", y sustituir x por el valor que se le asigna, que es $a+1$, con lo cual la precondición que obtenemos es "$a+1 = 0$":

$$\{a+1 = 0\}(\texttt{x:=a+1})\{x = 0\}$$

Ejemplo 4.7

$$\vdash \{x+y = 7\}(\texttt{z:=x+y})\{z = 7\}$$

Ejemplo 4.8

$$\vdash \{x+1 = 4\}(\texttt{x:=x+1})\{x = 4\}$$

Como se ve en estos ejemplos, la precondición se obtiene tomando la postcondición y sustituyendo en ella la variable por el valor que se le asigna.

4.4.2 Regla del condicional

La regla del condicional nos dice que si tenemos que demostrar una terna de la forma

$$\{p\}(\texttt{if } B \texttt{ then } (S_1) \texttt{ else } (S_2))\{q\}$$

podemos hacerlo en dos pasos: por un lado demostramos que, cuando partimos de la condición p y B es cierto, la ejecución de S_1 garantiza la condición q,

$$\{p \wedge B\}(S_1)\{q\}$$

y por otro lado, demostramos que, cuando partimos de la misma condición p y B es falso, la ejecución de S_2 también garantiza la condición q,

$$\{p \wedge \neg B\}(S_2)\{q\}$$

Ejemplo 4.9 Para demostrar que

$$\{x \neq 0\}(\texttt{if (x > 0) then (y:=x) else (y:=-x)})\{y > 0\}$$

basta demostrar que

$$\{x \neq 0 \wedge x > 0\}(\texttt{y:=x})\{y > 0\}$$

y que

$$\{x \neq 0 \wedge \neg(x > 0)\}(\texttt{y:=-x})\{y > 0\}$$

Naturalmente, al traducir las expresiones booleanas del programa al lenguaje de la lógica hay que recordar la equivalencia entre los operadores de programa y los operadores lógico-matemáticos. Por ejemplo, las expresiones booleanas `(x!=0)` y `(0=<x & x=<10)` se traducen, respectivamente, como $(x \neq 0)$ y $(0 \leq x \wedge x \leq 10)$.[7]

Como hemos visto, la regla del condicional es muy fácil de entender. Vamos a ver a continuación otra versión de esta regla que, aunque no es tan intuitiva ni tan fácil de recordar, resulta más cómoda de aplicar en la práctica.

4.4.3 Regla del condicional modificada

Queremos que el sistema, después de ejecutar la instrucción "`if` B `then (`S_1`) else (`S_2`)`" satisfaga la condición q.

$$\vdash \{p\}(\texttt{if } B \texttt{ then } (S_1) \texttt{ else } (S_2))\{q\}$$

Supongamos que hemos encontrado dos condiciones, p_1 y p_2, tales que $\{p_1\}(S_1)\{q\}$ y $\{p_2\}(S_2)\{q\}$. Definimos p así

$$p = (B \rightarrow p_1) \wedge (\neg B \rightarrow p_2) \tag{4.10}$$

Como $p \wedge B \rightarrow p_1$, por la regla de encadenamiento tenemos que

$$\{p \wedge B\}(S_1)\{q\}$$

Análogamente, como $p \wedge \neg B \rightarrow p_2$,

$$\{p \wedge \neg B\}(S_2)\{q\}$$

Introduciendo estos dos resultados en la regla del condicional (expresión (4.6)) tenemos que

$$\frac{\vdash \{p_1\}(S_1)\{q\} \qquad \vdash \{p_2\}(S_2)\{q\}}{\vdash \{(B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)\}(\texttt{if } B \texttt{ then } (S_1) \texttt{ else } (S_2))\{q\}} \tag{4.11}$$

Esta nueva versión de la regla del condicional nos dice que para asegurar que el sistema, después de ejecutar la instrucción "`if` B `then (`S_1`) else (`S_2`)`" satisfaga la postcondición q, es suficiente que satisfaga la precondición $(B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)$.

Esta nueva versión tiene la ventaja de que es más fácil buscar p_1 y p_2 por separado que tener que buscar directamente una condición p que satisfaga las dos condiciones, $\{p \wedge B\}(S_1)\{q\}$ y $\{p \wedge \neg B\}(S_2)\{q\}$. O visto de otra forma, el buscar p_1 y p_2 por separado es una forma de buscar p, de acuerdo con la ecuación (4.10).

En la sección 4.5.3 veremos cómo se aplica esta regla en la práctica.

[7] Tenga cuidado de no confundir "`x!=0`", que significa x es distinto de 0, con "$x! = 0$", que significa que el factorial de x es 0.

4.4.4 Regla del bucle

La regla del bucle está íntimamente ligada al concepto de invariante.

Definición 4.10 La condición p es un *invariante* para la instrucción "`while` B `do` (S)" si y sólo si se cumple que

$$\{p \wedge B\}(S)\{p\} \tag{4.12}$$

De esta definición se deduce que, si p es cierto antes de ejecutar la instrucción `while` también lo será después de haberla ejecutado. ¿Por qué? Supongamos que B es falsa. Entonces el cuerpo S no se ejecuta; por tanto, el estado del sistema no se modifica y la condición p sigue siendo cierta. Supongamos que B es cierta. Eso significa que el cuerpo S se ejecuta en un estado que satisface la condición $p \wedge B$, y entonces la propiedad (4.12) nos garantiza que después de ejecutar S el sistema va a seguir cumpliendo la condición p. Dicho de otro modo, no sabemos a priori cuántas veces se va a ejecutar el cuerpo S dentro de la instrucción `while`, pero sí sabemos que en cada una de las ejecuciones se va a mantener la propiedad p. Por eso p se denomina invariante. Esto es lo que nos permite concluir que

$$\frac{\vdash \{p \wedge B\}(S)\{p\}}{\vdash \{p\}(\texttt{while } B \texttt{ do } (S))\{p\}}$$

Por otro lado, la instrucción `while` sólo termina cuando B es falsa. Uniendo estos dos resultados tenemos

$$\frac{\vdash \{p \wedge B\}(S)\{p\}}{\vdash \{p\}(\texttt{while } B \texttt{ do } (S))\{p \wedge \neg B\}}$$

que es precisamente la regla del bucle.

Proposición 4.11 Para toda instrucción `while`, $\top$ es un invariante.

Demostración. La fórmula $\{\top \wedge B\}(S)\{\top\}$ es una tautología, porque la postcondición siempre es cierta. □

Naturalmente, en la práctica nos interesa encontrar invariantes no triviales. Veamos otros ejemplos de invariantes.

Ejemplo 4.12 La regla del bucle nos dice que para demostrar

$$\{x = 0\}(\texttt{while (y!=0) do (z:=3)})\{x = 0 \wedge \neg(y \neq 0)\}$$

basta demostrar que $\{x = 0 \wedge y \neq 0\}(\texttt{z:=3})\{x = 0\}$. (Esta fórmula se demuestra por la regla de asignación, pues en la postcondición "$x = 0$" no aparece z, y por eso al sustituir z por 3 la precondición que se obtiene es la misma, "$x = 0$".) El invariante es $x = 0$. □

En este ejemplo se da una paradoja: inicalmente no sabemos cuál es el valor de y y sin embargo concluimos que, después de ejecutar el bucle, se cumple que $\neg(y \neq 0)$, es decir, $y = 0$, a pesar de que el bucle no ha modificado el valor de y. ¿Cómo se explica esto? Debemos tener en cuenta dos situaciones: si inicialmente $y = 0$, la instrucción `while` no hace nada, y el programa termina, satisfaciendo la condición $\{x = 0 \wedge y = 0\}$. En cambio, si inicialmente $y \neq 0$, el programa entra en un bucle infinito. Por eso no hay contradicción al afirmar que "si el programa termina (algo que no ocurre cuando $y \neq 0$), el valor de y al salir del bucle es 0".

Como ya hemos mencionado, el sistema deductivo de Hoare sólo garantiza la corrección parcial: la afirmación $\vdash \{p\}(S)\{q\}$ es equivalente a $\vDash_{\text{par}} \{p\}(S)\{q\}$, que, como vimos anteriormente, significa que "si el sistema satisface inicialmente la condición p y ejecuta el programa S **y el programa termina**, entonces el sistema satisface la condición q".

En el ejemplo anterior era muy fácil encontrar un invariante: como el cuerpo de la instrucción `while`, que es `z:=3`, no modifica el valor de x ni de y, cualquier condición en que sólo aparezcan estas dos variables será un invariante.

Otro ejemplo similar es el siguiente:

Ejemplo 4.13

$$\vdash \{x < 5\}(\texttt{while (x==0) do (y:=1)})\{x < 5 \wedge \neg(x = 0)\}$$

En estos dos ejemplos, el cuerpo de la instrucción `while`, S, no modifica la condición, B, y por eso hay dos posibilidades: o bien la condición es cierta antes de ejecutar la instrucción `while`, con lo cual el programa entra en un bucle infinito, o bien la condición es falsa antes de ejecutar la instrucción, con lo cual la instrucción `while` no hace nada, y es como si no estuviera en el programa. Por eso en la práctica sólo nos interesan los casos en que el cuerpo, S, puede modificar la condición, B.

Ejemplo 4.14 Queremos demostrar que

$$\vdash \{x \geq 0\}(\texttt{while (x!=0) do (x:=x-1)})\{x = 0\}$$

Para ello, demostramos la expresión

$$\vdash \{x \geq 0 \wedge x \neq 0\}(\texttt{x:=x-1})\{x \geq 0\}$$

que indica que "$x \geq 0$" es un invariante para este bucle. Aplicando la regla del bucle, con las equivalencias $p =$ "$x \geq 0$", $B =$ "$x \neq 0$" y $S =$ "`x:=x-1`" se obtiene que

$$\vdash \{x \geq 0\}(\texttt{while (x!=0) do (x:=x-1)})\{x \geq 0 \wedge \neg(x \neq 0)\}$$

de donde se deduce la expresión que queríamos demostrar.

4.4.5 Regla de composición

La regla de composición es muy sencilla: si queremos demostrar $\{p\}(S_1;S_2)\{r\}$ tenemos que buscar una propiedad q tal que $\{p\}(S_1)\{q\}$ y $\{q\}(S_1;S_2)\{r\}$.

Ejemplo 4.15 Para demostrar que

$$\{x \geq 0\}(\texttt{y:=x+1;z:=2*y})\{z \geq 0\}$$

basta demostrar estas dos fórmulas:

$$\{x \geq 0\}(\texttt{y:=x+1})\{y \geq 0\}$$
$$\{y \geq 0\}(\texttt{z:=2*y})\{z \geq 0\}$$

o bien estas dos:

$$\{x \geq 0\}(\texttt{y:=x+1})\{y \geq 1\}$$
$$\{y \geq 1\}(\texttt{z:=2*y})\{z \geq 0\}$$

4.4.6 Regla de encadenamiento

La regla de encadenamiento también es muy sencilla y ya la hemos aplicado en algunos de los ejemplos anteriores. Tan sólo vamos a indicar que de la regla de encadenamiento y de la propiedad $p \rightarrow p$ se deducen estas dos reglas, que pueden considerarse como casos particulares de ella:

$$\frac{\vdash \{p \rightarrow p'\} \qquad \vdash \{p'\}(S)\{q\}}{\vdash \{p\}(S)\{q\}}$$

$$\frac{\vdash \{p\}(S)\{q'\} \qquad \vdash \{q' \rightarrow q\}}{\vdash \{p\}(S)\{q\}}$$

4.5 Verificación parcial de programas

4.5.1 Composición y encadenamiento

Supongamos que tenemos un programa S compuesto por una serie de instrucciones, $S = S_1; \ldots; S_n$. Una forma de expresar la verificación de este programa consiste en escribir una cadena de condiciones e instrucciones, de este modo,

$\{p\}$
S_1
$\{p_1\}$ [Justificación-1]
$\vdots$
$\{p_{n-1}\}$ [Justificación-$(n-1)$]
S_n
$\{q\}$ [Justificación-n]

(Definimos $p_0 = p$ y $p_n = q$.) Para cada eslabón $\vdash \{p_{i-1}\}(S_i)\{p_i\}$ se indica entre corchetes cuál es la regla que lo justifica. Al demostrar todos y cada uno de los eslabones, el programa S queda demostrado, por la regla de composición. Si S_i es una instrucción compuesta, hay que repetir el proceso anterior anidando las cadenas de demostración, y así sucesivamente, hasta llegar a instrucciones simples.

A veces puede resultar que la postcondición de S_i no coincida con la precondición que requiere S_{i+1}. En ese caso es posible insertar varias condiones entre S_i y S_{i+1},

$\{p\}$
S_i
$\{p_{i,1}\}$ [Justificación-i]
$\{p_{i,2}\}$ [Justificación-i,1]
$\vdots$
$\{p_{i,k}\}$ [Justificación-i,k]
S_{i+1}

de modo que la primera condición intermedia, $p_{i,1}$, es la postcondición de S_i, y la última condición intermedia, $p_{i,k}$, sirve de precondición para S_{i+1}. Para cada par de proposiciones consecutivas, la fórmula $\vdash p_{i,j} \rightarrow p_{i,j+1}$ se demuestra por alguno de los axiomas del dominio o por alguna de las propiedades de la lógica de predicados, que se indica entre corchetes como [Justificación-i,j]. La validez de la demostración viene garantizada por la regla de encadenamiento.

Observe que en el proceso de verificación estamos combinando dos sistemas deductivos. Por un lado, tenemos el sistema deductivo de Hoare, que nos permite justificar los eslabones del tipo $\vdash \{p_{i-1}\}(S_i)\{p_i\}$, y por otro lado el sistema deductivo propio del dominio, que nos permite justificar los eslabones del tipo $\vdash p_{i,j} \rightarrow p_{i,j+1}$. Como hemos dicho ya, en el caso de nuestro micro-lenguaje, el dominio es la aritmética de los números enteros.

Ejemplo 4.16 La demostración de $\vdash \{\top\}$(`x:=2;y:=3*x+1`)$\{y = 7\}$ puede ser ésta:

$\{\top\}$
$\{2 = 2\}$ [$\forall x,\ x = x$]
`x:=2`
$\{x = 2\}$ [Regla de asignación]
$\{3x + 1 = 7\}$ [$3 * 2 + 1 = 7$]
`y:=3*x+1`
$\{y = 7\}$ [Regla de asignación]

Observe que la demostración de $\vdash \{2 = 2\}$(`x:=2`)$\{x = 2\}$ y de $\vdash \{3x + 1 = 7\}$(`y:=3*x+1`)$\{y = 7\}$ se basa en la regla de asignación, que pertenece al sistema deductivo de Hoare, mientras que $\vdash \top \rightarrow$ 2=2 y $\vdash (x = 2) \rightarrow (3x + 1 = 7)$ se demuestran mediante el sistema deductivo de la aritmética. □

Aunque resulta más intuitivo entender la demostración leyéndola en el sentido de ejecución del programa, es decir, de arriba a abajo (o hacia adelante, si se prefiere), la construcción de la demostración suele ser más sencilla en sentido inverso, es decir, desde la postcondición hasta la precondición (de abajo a arriba, o hacia atrás). La razón es que generalmente la regla de asignación y la del condicional son más fáciles de aplicar hacia atrás que hacia adelante, como vamos a ver a continuación.

4.5.2 Tratamiento de las asignaciones

La regla de asignación puede expresarse así:

$\{P(E)\}$
`x := E`
$\{P(x)\}$ [Regla de asignación]

Como acabamos de indicar, generalmente esta regla es más fácil de aplicar hacia atrás. Lo vemos volviendo al ejemplo anterior.

Ejemplo 4.17 Queremos demostrar que $\vdash \{\top\}$(`x:=2;y:=3*x+1`)$\{y = 7\}$. Para ello escribimos la precondición, las instrucciones simples y la postcondición. Entre cada par de instrucciones insertamos la condición correspondiente que, como aún no conocemos, la hemos representado mediante una interrogación. Una interrogación entre corchetes indica que aún no hemos demostrado ese paso. (Si el lector lo prefiere, al construir sus demostraciones puede dejar un espacio en blanco en vez de escribir una interrogación; nosotros utilizamos la interrogación para que quede más claro cuándo hay todavía algún paso pendiente de demostrar.)

$\{\top\}$
`x := 2`
$\{?\}$ [?]
`y := 3*x+1`
$\{y = 7\}$ [?]

Si queremos construir la demostración de arriba a abajo, tenemos que buscar un predicado P que nos permita aplicar la regla de asignación $\{P(E)\}$`(x:=2)`$\{P(x)\}$. El problema es que, en nuestro esbozo de demostración, la precondición de `x:=2` es $\top$; si tomamos este predicado tenemos que $P(E) = P(2) = \top$, con lo que concluimos que $\vdash \{\top\}$`(x:=2)`$\{\top\}$, lo cual es cierto pero no nos sirve para nada. Para esta primera instrucción tampoco podemos buscar el predicado P a partir de su postcondición porque aún no la conocemos (por eso hemos escrito una interrogación).

Vamos a intentar verificar el programa anterior de abajo a arriba. Ahora sí es fácil aplicar la regla de asignación: basta tomar la postcondición de la última instrucción, $y = 7$, y en esta expresión sustituimos y por el valor asignado, con lo cual obtenemos la precondición de `y:=3*x+1`:

$\{\top\}$
`x := 2`
$\{3 * x + 1 = 7\}$ [?]
`y := 3*x+1`
$\{y = 7\}$ [Regla de asignación]

Siguiendo el mismo proceso para la instrucción `x := 2`, tomamos su postcondición y sustituimos x por el valor asignado: con lo cual llegamos a

$\{\top\}$
$\{3 * 2 + 1 = 7\}$ [?]
`x := 2`
$\{3 * x + 1 = 7\}$ [Regla de asignación]
`y := 3*x+1`
$\{y = 7\}$ [Regla de asignación]

Para concluir la demostración, basta probar que $\top \rightarrow 3 * 2 + 1 = 7$, que es equivalente a probar que $3 * 2 + 1 = 7$ (habría que demostrarlo por la aritmética de números enteros). □

4.5.3 Tratamiento de las instrucciones condicionales

Para tratar las instrucciones `if-then-else` aplicamos la regla del condicional modificada (expresión (4.11)), que se traduce en:

$\{(B \rightarrow p_1) \wedge (\neg B \rightarrow p_2)\}$
`if` B `then (`
$\{p_1\}$
S_1
$\{q\}$ [Justificación-1]
`) else (`
$\{p_2\}$
S_2
$\{q\}$ [Justificación-2]
`)`
$\{q\}$ [Regla del condicional]

Ejemplo 4.18 Para demostrar que

$$\{x \neq 0\}\texttt{(if (x>0) then (y:=x) else (y:=-x))}\{y > 0\}$$

escribimos este esbozo de demostración:

$\{x \neq 0\}$
$\{?\}$ [?]
`if (x>0) then (`
$\{p_1?\}$
`y:=x`
$\{y > 0\}$ [?]
`) else (`
$\{p_2?\}$
`y:=-x`
$\{y > 0\}$ [?]
`)`
$\{y > 0\}$ [?]

Las condiciones p_1 y p_2 se obtienen por la regla de asignación:

$\{x \neq 0\}$
$\{?\}$ [?]
`if (x>0) then (`
$\{x > 0\}$
`y:=x`
$\{y > 0\}$ [Regla de asignación]
`) else (`
$\{(-x) > 0\}$
`y:=-x`
$\{y > 0\}$ [Regla de asignación]
`)`
$\{y > 0\}$ [?]

La precondición de la instrucción condicional es $(B \to p_1) \wedge (\neg B \to p_2)$, de modo que

$\{x \neq 0\}$
$\{(x > 0 \to x > 0) \wedge (\neg(x > 0) \to (-x) > 0\}$ [?]
`if (x>0) then (`
$\{x > 0\}$
`y:=x`
$\{y > 0\}$ [Regla de asignación]
`) else (`
$\{(-x) > 0\}$
`y:=-x`
$\{y > 0\}$ [Regla de asignación]
`)`
$\{y > 0\}$ [Regla de asignación]

Dejamos como ejercicio para el lector demostrar que $x \neq 0 \to (x > 0 \to x > 0) \wedge (\neg(x > 0) \to (-x) > 0$, con lo cual se completa la verificación del programa. (En este caso, la verificación parcial es una verificación total, porque el programa no contiene bucles.)

4.5.4 Tratamiento de los bucles

Hemos visto anteriormente que el tratamiento de las instrucciones de asignación y condicionales es bastante fácil, pues se limita a la aplicación de un algoritmo. Sin embargo, el tratamiento de los bucles es mucho más complicado, pues requiere encontrar invariantes, una tarea para la cual no existe un algoritmo, sino tan sólo algunas heurísticas.[8]

Cuando en la demostración de un programa nos encontramos con una situación como ésta,

$\{?\}$
`while` B `do (`S`)`
$\{q\}$ [?]

tenemos que encontrar un invariante que nos permita escribir

$\{p\}$
`while` B `do (`
$\{p \wedge B\}$
S
$\{p\}$ [Justificación (de que p es un invariante)]
`)`
$\{p \wedge \neg B\}$ [Regla del bucle]
$\{q\}$ $[p \wedge \neg B \rightarrow q]$

Para ello, el invariante p debe satisfacer tres condiciones:

1. $\vdash \{p \wedge B\}(S)\{p\}$ (es la definición de invariante);
2. $\vdash p \wedge \neg B \rightarrow q$ (para poder obtener la postcondición del bucle);
3. Hace falta que p pueda deducirse a partir de la postcondición de la instrucción anterior al bucle (para poder continuar luego la demostración hacia arriba).

Ejemplo 4.19 Dado el programa `fact1` introducido en el ejemplo 4.1, queremos demostrar que se cumple la condición (4.1).[9] Para ello buscamos un invariante p que cumpla que $\vdash \{p \wedge i \neq x\}$(`i:=i+1;fact:=fact*i;`)$\{p\}$ y $\vdash p \wedge \neg(i \neq x) \rightarrow fact = x!$.

Una heurística que recomiendan algunos autores para encontran invariantes es construir una tabla que refleje el valor que toman las variables del programa en cada ejecución del bucle, y tratar de ver qué propiedad(es) se cumplen para cada una de las filas. Para este ejemplo, tomando $x = 6$ (un valor escogido arbitrariamente, con la única condición de que no sea demasiado grande ni demasiado pequeño), obtenemos la tabla 4.1. En todas las columnas de esta tabla se cumple que $fact = i!$. Vamos a comprobar si esta condición es un invariante:

[8] Como ya sabe el lector, en inteligencia artificial se denomina *heurística* a una regla que ayuda a buscar una solución, aunque generalmente la aplicación de una heurística ni garantiza que se encuentre una solución, ni garantiza que la solución encontrada sea óptima.

Por eso podríamos decir que el tratamiento de las condiciones de asignación y condicionales es una técnica, mientras que el tratamiento de los bucles es un arte, que sólo se aprende con la práctica.

[9] Recordemos que la especificación (4.2) era más correcta, pero la especificación (4.1) también era válida cuando el programa no modifica el valor de la variable `x`, y en nuestro caso es más sencilla de aplicar. Dejamos como ejercicio para el lector comprobar que el programa `fact1` cumple la especificación (4.2).

iteración	x	i	$fact$	B
1^a	6	0	1	$\top$
2^a	6	1	1	$\top$
3^a	6	2	2	$\top$
4^a	6	3	6	$\top$
5^a	6	4	24	$\top$
6^a	6	5	120	$\top$
7^a	6	6	720	$\bot$

Tabla 4.1: Tabla para probar el bucle de la función `fact1`..

$\{fact = i! \wedge i \neq x\}$
```
i := i + 1;
```
$\{?\}$ [?]
```
fact := fact * i;
```
$\{fact = i!\}$ [?]

Aplicando dos veces la regla de asignación llegamos a

$\{fact = i! \wedge i \neq x\}$
$\{fact = i!\}$ $[p \wedge q \rightarrow p]$
$\{fact * (i+1) = (i+1)!\}$ $[\forall a,\ (a+1)! = (a+1) * a!]$
```
i := i + 1;
```
$\{fact * i = i!\}$ [Regla de asignación]
```
fact := fact * i;
```
$\{fact = i!\}$ [Regla de asignación]

lo cual demuestra que $fact = i!$ es un invariante. Podemos integrar este resultado en la verificación del programa completo, que queda así:

$\{x \geq 0\}$
$\{1 = 0!\}$ [Axioma (definición de factorial)]
```
fact := 1;
```
[Regla de asignación]
$\{fact = 0!\}$
```
i := 0;
```
$\{fact = i!\}$ [Regla de asignación]
```
while (i != x) do (
```
$\{fact = i! \wedge i \neq x\}$
$\{fact = i!\}$ $[p \wedge q \rightarrow p]$
$\{fact * (i+1) = (i+1)!\}$ $[\forall a,\ (a+1)! = (a+1) * a!]$
```
    i = i + 1;
```
$\{fact * i = i!\}$ [Regla de asignación]
```
    fact := fact * i;
```
$\{fact = i!\}$ [Regla de asignación]
```
)
```
$\{fact = i! \wedge \neg(i \neq x)\}$ [Regla del bucle]
$\{fact = i! \wedge i = x\}$ $[\forall a, \forall b,\ a \neq b \leftrightarrow \neg(a = b)]$
$\{fact = x!\}$ [Regla de sustitución (propiedad de la lógica)]

Observe que la precondición $\{x \geq 0\}$ no nos ha hecho falta para demostrar $\{0! = 1\}$, porque éste es un axioma del sistema. Por tanto, también habríamos podido demostrar que

$$\vdash \{\top\}(\texttt{fact1})\{fact = x!\}$$

Sin embargo, cuando $x < 0$, el programa `fact1` entra en un bucle infinito. Por eso la precondición $\{x \geq 0\}$ es necesaria para garantizar la correción total, mientras que, de acuerdo con la expresión que acabamos de escribir, no es necesaria ninguna precondición para garantizar la correción parcial.

Ejercicio 4.20 Demostrar que

$$\vdash \forall a,\ \{x = a\}(\texttt{fact1})\{fact = a!\}$$
$$\vdash \forall a,\ \{x = a\}(\texttt{fact2})\{fact = a!\}$$

4.6 Verificación total de programas

El sistema deductivo expuesto en la sección 4.4 sólo garantiza la corrección parcial. Para demostrar la corrección total es necesario sustituir la regla del bucle anterior por esta otra:[10]

Regla del bucle (para corrección total):

$$\frac{\vdash \{p \wedge B \wedge E \geq 0 \wedge E = a\}(S)\{p \wedge E \geq 0 \wedge E < a\}}{\vdash \{p \wedge E \geq 0\}(\texttt{while } B \texttt{ do } (S))\{p \wedge \neg B\}}$$

En esta regla p sigue siendo un invariante, porque si se cumple antes de que se ejecute el bucle, se cumple también después. Si hemos demostrado ya la corrección parcial para este bucle, podemos utilizar aquí el mismo invariante.

La expresión E se denomina *variante*,[11] porque si es igual a a (un número entero) antes de que se ejecute el bucle, será estrictamente menor que a cuando el bucle se ha ejecutado, y continuará decreciendo cada vez que se ejecuta el cuerpo del bucle. Como esta expresión no puede decrecer indefinidamente (porque siempre es mayor que 0), el bucle debe terminar.

De este modo, el nuevo sistema deductivo garantiza tanto la corrección parcial como la terminación de los bucles, con lo que se demuestra la corrección total.

Ejemplo 4.21 Dado el programa `fact1` introducido en el ejemplo 4.1, queremos demostrar que se cumple la condición (4.1) en el nuevo sistema deductivo (que garantiza la corrección total). Recordemos que el bucle que aparece en este programa es

```
while (i!=x) do (i=i+1; fact:=fact*i)
```

y que en la verificación parcial habíamos tomado como invariante $fact = i!$. Por tanto, aunque todavía no hemos encontrado E, sabemos que se cumple que

$$\vdash \{p \wedge B \wedge E \geq 0 \wedge E = a\}(S)\{p\}$$

[10] Desde un punto de vista conceptual, no era necesario definir primero el sistema deductivo de verificación parcial, sino que podríamos haber definido directamente el de verificación total. Sin embargo, por motivos pedagógicos nos ha parecido mejor abordar el problema en dos pasos, discutiendo primero la verificación parcial y viendo luego cómo se debe modificar la regla del bucle para obtener la verificación total.

[11] Observe que el invariante es *una proposición* (por ejemplo, $fact = i!$), y por eso lo hemos representado por p, mientras que el variante es *una expresión numérica* (por ejemplo, $x - i$), y por eso lo representamos por E. Observe también que, en general, tanto p como E hacen referencia al valor de las variables del programa.

es decir,

$$\vdash \{fact = i! \wedge i \neq x \wedge E \geq 0 \wedge E = a\}(\texttt{i:=i+1;fact:=fact*i})\{fact = i!\}$$

(lo hemos demostrado en el ejemplo 4.19).

Vamos a buscar ahora una expresión E (un variante) que cumpla que

$$\vdash \{p \wedge B \wedge E \geq 0 \wedge E = a\}(S)\{E \geq 0 \wedge E < a\}$$

Tomamos la expresión $x - i$, pues cumple que

$$\vdash \{i \neq x \wedge x - i \geq 0 \wedge x - i = a\}(\texttt{i:=i+1;fact:=fact*i})\{x - i \geq 0 \wedge x - i < a\}$$

(Invitamos al lector a que lo demuestre formalmente.) Esto implica que si antes de ejecutar el cuerpo del bucle se cumple $0 \leq x - i = a$, después de su ejecución se cumple $0 \leq x - i < a$. Como la expresión E siempre decrece al menos una unidad en cada ejecución del cuerpo del bucle (en este ejemplo $x - i$ decrece exactamente una unidad) y siempre es mayor o igual que 0, el bucle debe terminar. □

Ejercicio 4.22 Demostrar la corrección total de `fact2` dada la especificación (4.2). □

Como en el caso de los invariantes, encontrar un variante puede ser complicado, ya que no hay reglas algorítmicas, sino sólo algunos consejos heurísticos. Por ejemplo, se recomienda la construcción de una tabla de ejecución, como la del ejemplo 4.19 (tabla 4.1), que puede ayudar a encontrar tanto el invariante como el variante.

4.7 Comentarios adicionales

4.7.1 Consistencia y completitud

El sistema deductivo que hemos definido en la sección 4.4 es **consistente** para la verificación parcial, lo cual significa que toda terna deducida mediante la aplicación de sus reglas es parcialmente correcta:

$$\vdash \{p\}(S)\{q\} \text{ implica que } \models_{\text{par}} \{p\}(S)\{q\}$$

Recordemos que $\models_{\text{par}} \{p\}(S)\{q\}$ que significa que "si el sistema (el ordenador) se encuentra inicialmente en alguno de los estados representados por la precondición p y ejecuta el programa S y el programa termina, el sistema se encontrará en alguno de los estados representados por la postcondición q" (cf. sec. 4.3.2).

Análogamente, el sistema deductivo que hemos definido en la sección 4.6 es **consistente** para la verificación total:

$$\vdash \{p\}(S)\{q\} \text{ implica que } \models_{\text{tot}} \{p\}(S)\{q\}$$

Por tanto, la consistencia significa que un programa verificado mediante este sistema deductivo satisface realmente la especificación $\{p\}(S)\{q\}$.

Para demostrar la consistencia de ambos sistemas deductivos deberíamos establecer primero una semántica formal de nuestro micro-lenguaje de programación (cómo y cuándo se pasa de un estado a otro al ejecutar cada una de las instrucciones del programa), algo que no hemos hecho. Sin embargo, el conocimiento que tiene el lector sobre cómo funciona cada una de esas instrucciones en un lenguaje de programación y las explicaciones que hemos dado para justificar cada regla hacen verosímil la afirmación de que el sistema es consistente. El lector interesado en la demostración formal de la consistencia puede consultar el libro de Francez [1992] o el de Apt y Olderog [1997].

A su vez, cada uno de estos sistemas es **completo en sentido relativo**. La completitud es la propiedad recíproca de la consistencia, y significa que toda terna semánticamente correcta puede ser obtenida mediante el sistema deductivo correspondiente: si se cumple $\models_{\text{tot}} \{p\}(S)\{q\}$ entonces esta terna puede ser demostrada (el programa puede ser verificado) por el sistema deductivo de verificación total:

$$\models_{\text{tot}} \{p\}(S)\{q\} \text{ implica que } \vdash \{p\}(S)\{q\}$$

La propiedad de completitud para el sistema deductivo de verificación parcial es análoga.

La expresión "completo **en sentido relativo**" significa que completo si consideramos cada fórmula del dominio como un axioma (recordemos que en el caso de nuestro micro-lenguaje el dominio es la aritmética de números enteros). En la práctica es imposible incluir todas las fórmulas del dominio dentro del sistema deductivo, porque son infinitas. Por eso nos interesa que las fórmulas del dominio puedan deducirse a partir de un número finito de axiomas. Sin embargo, no se puede construir un sistema deductivo completo para la aritmética de números enteros. Dado que el sistema deductivo de verificación que hemos tratado en este capítulo (en sus dos versiones, parcial y total), además de tener como axiomas las cinco reglas propias de la verificación, incluye también los axiomas del sistema deductivo del dominio, si éste no es completo tampoco puede serlo aquél. Por eso se dice que el sistema deductivo de verificación de programas **no es completo en sentido absoluto**.

4.7.2 Otras cuestiones

El micro-lenguaje que hemos estudiado en este capítulo está limitado sobre todo porque no utiliza arrays ni procedimientos, dos recursos imprescindibles para desarrollar programas de cierta envergadura. Estos aspectos se estudian en el libro de Francez [1992].

Según los métodos presentados en este capítulo, antes de verificar un programa es necesario haberlo construido. Sin embargo, es posible integrar ambas tareas en una sola, si se escribe el programa mediante bloques que se enlazan y expanden; en cada uno de los pasos se verifica el código generado, hasta llegar a un programa cuya corrección está garantizada. Puede encontrarse más información sobre este método en el libro de Apt y Olderog [1997].

Bibliografía complementaria

La referencia histórica básica para este tema es [Hoare, 1969]. Los libros de de Huth y Ryan y de Ben-Ari que hemos mencionado en la Bibliografía Recomendada *(pág. 3) explican este tema, aunque dejan algunos cabos sueltos (especialmente el libro de Ben-Ari). Un tratamiento mucho más completo y riguroso se encuentra en dos libros excelentes: [Apt y Olderog, 1997] y [Francez, 1992].*

Por otro lado, los recursos bibliográficos e informáticos, disponibles en Internet, que hemos comentado en la sección Motivación para los alumnos de Ingeniería Informática *(pág. 2) pueden ser útiles para hacer más interesante el estudio de este tema.*

Actividades y evaluación

Los ejercicios de evaluación más importantes de este tema consisten, naturalmente, en la verificación de pequeños programas secuenciales mediante la lógica de Hoare; por ejemplo, los ejercicios 7 a 10 del capítulo 9 del libro de texto [Ben-Ari, 2001, pág. 220], o los ejercicios que aparecen al final de cada sección en [Huth y Ryan, 2000, cap. 4].