**Capítulo 15**

**Desarrollo de bucles a partir de invariantes y funciones variante**

En este capítulo se analizan dos métodos para desarrollar un bucle cuando se dan la precondición ***P,*** la postcondición ***Q,*** el invariante ***I*** y la función variante *t*. El primer método conduce naturalmente a un bucle con un solo comando guardado, **do *B -> S* od.**  El segundo aprovecha la flexibilidad de la construcción iterativa y generalmente da como resultado bucles con más de un comando protegido.

La lista de verificación 11.9 se usará mucho, y puede ser prudente revisarla antes de continuar. Como es nuestra práctica en todo momento, las partes del desarrollo que ilustran los principios que deben cubrirse se discuten de manera formal y detallada, mientras que otras partes se tratan de manera más­ indirecta.

**15.1 Desarrollando el guardian primero**

*sumando los elementos de una matriz*

Considere el siguiente problema. Escribe un programa que, dado el entero *n* >=0 y la matriz entera fijab [0: n-1]**,** almacene en la variable ***s*** la suma de los elementos de *b.*  La condición previa ***P*** es simplemente *n >= 0;* la postcondición ***Q*** es

*Q:*

Se desea un bucle con el siguiente invariante y función variante.

I:

T: n-i

Así, se ha introducido la variable *i*. **El invariante establece que en cualquier punto del cálculo *s* contiene la suma de los primeros valores de *b.***

La asignación *i, s:=* 0, 0 obviamente establece *I,* por lo que será suficiente como inicialización. (Tenga en cuenta que *i, s:= 1, b[0]* no es suficiente porque, si *n = 0,* no se puede ejecutar. Si *n* = 0, la ejecución del programa debe establecer *s* en la identidad de adición, 0.)

El siguiente paso es determinar la guardia *B* para el bucle **do** *B -> S* **od.**

La lista de verificación 11.9 requiere *,* por lo que se elige para satisfacerla. Comparando­ *I* y *Q,* concluimos que *i = n* servirá. La guardia *B* deseada del bucle es, por lo tanto, su complemento, El programa se ve como

*i, s: =* 0, 0; **do od**

Ahora para el comando. El propósito del comando es progresar hacia la terminación —es decir, disminuir la función enlazada *t—* y una primera opción obvia para ello es *i :=* i+1. Pero, esto destruiría el invariante­, y para restablecerla *b[i]* debe agregarse simultáneamente a *s.* Por lo tanto, el programa es

(15.1.1) *i, s: =* 0, 0; **do od**

**Observación:** Para aquellos que se sienten incómodos con la asignación múltiple, la prueba formal de que *I* se mantiene es la siguiente. Tenemos

*WP ("i, s:= i+1, s+b[i]",I)*

y esto es implicado por **❑**

***Discusión***

En primer lugar, discutamos el equilibrio entre formalidad e intuición que se observa aquí. Las condiciones previas y posteriores, la función invariante y la función enlazada se dieron formal y precisamente. El desarrollo de las partes del programa se dio de manera menos formal, pero la lista de verificación 11.9, que se basa en el teorema formal para el comando iterativo, proporcionó la mayor parte de la motivación y la visión. Para comprobar el desarrollo informal, nos basamos en la teoría (al comprobar que el cuerpo del bucle mantenía el invariante­). Esto es ilustrativo del enfoque general (13.1) mencionado en el capítulo 13.

Una estrategia importante en el desarrollo fue encontrar al guardia antes del comando. Y la consideración principal al encontrar el guardia *B* era que tenía que satisfacer a *.* Por lo tanto,se desarrolló y luego­ se uso su complemento para producir *B.*

Algunos se oponen al principio a encontrar la guardia de esta manera, porque tradicionalmente usarían la guardia *i < n* en lugar de *.*  Sin embargo, es mejor, porque un error de software o hardware que hizo que *i > n* resultaría en una ejecución no terminante. Es mejor perder el tiempo de la computadora que sufrir las consecuencias de que un error pase desapercibido (es mejor que el ciclo no termine), lo que sucedería si se usara el guardia *i < n*. Este análisis conduce a lo siguiente

(15.1.2) **•Principio:** Todas las demás cosas son iguales, hacer que las guardias de un bucle sean lo más *débiles* posible, de modo que un error pueda causar un bucle infinito.

El principio 15.1.2 debe compararse con el principio 14.10, que se refiere a los guardias de un comando alternativo.

El método utilizado para desarrollar la guardia de un bucle es extremadamente simple­ y confiable, ya que se basa en la manipulación de expresiones matemáticas estáticas. En este sentido, recuerdo mis viejos tiempos de programación FORTRAN —principios de la década de 1960— cuando a veces se necesitaban tres ejecuciones de depuración ­para lograr la terminación adecuada del bucle. La primera vez el bucle iteró una vez muy pocos, la segunda vez demasiados y la tercera vez justo. Fue un proceso frustrante de prueba y error. Ya no es necesario; simplemente desarrolle para satisfacer y complementarlo.

Otro punto importante sobre el desarrollo fue el énfasis en la terminación. La necesidad de avanzar hacia la terminación motivó el desarrollo del cuerpo del bucle; restablecer el invariante fue la segunda consideración. En realidad, cada bucle con un comando protegido tiene la interpretación de alto nivel.

(15.1.3) *{invariante: I}*

*{función variante: t***}**

**do** Disminuir *t,* manteniendo *I true* **od**

*{* }

Este enfoque para el desarrollo de bucles se resume de la siguiente manera:

(15.1.4) **•Estrategia para desarrollar un bucle:** Primero desarrollar la guardia ***B*** para que *;*  luego desarrolle el cuerpo para que disminuya la función variante mientras restablece ­el bucle invariante.

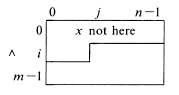
***Búsqueda de una matriz bidimensional***

Considere el siguiente problema.

Escriba un algoritmo que, dada una matriz b [0: m —1] [0: n-1]*,* donde 0 < m y 0 *< n,* busque en *b* un valor fijo *x.*  Si x ocurre en varios lugares de *b,* no importa qué lugar se encuentre. Para este problema, usaremos la notación bidimensional convencional, escribiendo *b* como *b [0: m-1*, 0: n-1]. Usando las variables *i* y *j,* al finalizar *x =* *b[i, j]* o, si esto no es posible, *i* = *m .*  Para ser más precisos, la ejecución del programa debe establecer

(15.1.5) *Q: .*

El invariante *I,* dado a continuación usando un diagrama, establece que x no está en las filas ya buscadas b [0:i-1] y no en las columnas ya buscadas b [i, 0:j-1]de la fila actual *i.*



|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | *x* no aquí | |
| (15.1.6) *I:* |  |  |  |
|  |  |  |  |
| *M-* |  |  |  |

La función variante t es el número de valores en la sección no probada: *(m-i)\*n - j .* Como primer paso en el desarrollo, antes de leer más a fondo, determine la inicialización del bucle.

La elección obvia es *i, j :=* 0, 0, porque entonces la sección en la que "x no está aquí" está vacía. A continuación, ¿cuál debería ser la guardia *B* del bucle?

La expresión,debe satisfacer*.* Debe ser lo suficientemente fuerte como para que se pueda establecer cada una de las dos disyuntivas de *Q*. Para proporcionar la primera disyunción, elija *i* < m **cand** *x = b [i, j];* para proporcionar el segundo, elija *i* = m. El operador **cand** es necesario para asegurar que la expresión está bien definida, ya que *b[i,j]* puede ser indefinido si . Por lo tanto, elija para ser

*: i = m V ( i < m* **cand x =** *b [i , j])* Usando las leyes de De Morgan, encontramos su complemento *B:*

*B:*

Ya que la guardia *B* debe evaluarse sólo cuando el invariante *I* es cierto, lo que significa que *i m* es cierto, se puede simplificar a:

*B:* y finalmente a

*B:*

La línea final es, por lo tanto, la guardia del bucle. El siguiente paso es construir el ­cuerpo del bucle. Hazlo, antes de seguir leyendo.

El propósito del cuerpo del bucle es disminuir la función enlazada *t,* que es el número de elementos en la sección no probada: *(m —i)\*n —j.,* la condición bajo la cual se ejecuta el cuerpo, implica que *i* < m, *j < n* y *,* de modo que el elemento *b[i, j],* que se encuentra en la sección no probada, se puede mover a la sección probada. Un comando posible para hacer esto es *j :=* j+1, pero mantiene el invariante *I* solo si *j < n-1.*  Así que tenemos el comando guardado

*j <n-1* ***->*** *j := j+1*

¿Qué hacemos si *?* En este caso, debido a que el invariante *I* es verdadero, tenemos . Por lo tanto, debemos determinar qué hacer si *j = n-1,* es decir, si *b [i, j]* es el elemento más a la derecha de su fila. Para mover *b [i, j]* a la sección probada se requiere pasar al principio de la siguiente fila, es decir, ejecutar .

El cuerpo del bucle es, por lo tanto,

**IF *FI***

El programa es, por lo tanto

(15.1.7) *i, j:=* 0, 0

**do** **IF**

**FI**

**od**

Si se desea, el cuerpo del bucle se puede reorganizar:

i, j := 0, 0

**do**  j := j + 1;

**IF**

**FI**

**od**

***Discusión***

Tenga en cuenta que la operación **cand** (en lugar de ) es realmente necesaria.

Tenga en cuenta que el método para desarrollar un comando alternativo se utilizó al desarrollar el cuerpo del bucle, aunque de manera informal. Primero, se eligió el com­ando *j := j+1*, y se vio que funcionaba como se deseaba solo si *j < n-1.*  Formalmente, uno debe probar

pero este caso es lo suficientemente simple como para manejarlo de manera informal, si se usa cuidado. En segundo lugar, se eligió el comando *i, j := i+1,* 0 para manejar el caso restante , *j = n.*

Nótese que el comando alternativo tiene los guardias j < n-1 y j = n - 1, y no y . Las guardas se han hecho lo más fuertes posible, de acuerdo con el principio **14.10,** con el fin de detectar errores.

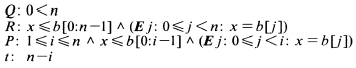
Desarrollaremos otra solución a este problema en la sección 15.2. **Ejercicios para la Sección 15.1 (P: invariante, Q: pre, R: post, t: enlazada)**

**1.** Desarrolle un segundo programa para el primer ejemplo de esta sección. Esta vez utilice la función invariante y enlazada

*P: 0 --\_,i --<... n* A *S =(//:i-<.,i<n:bUD t: i*

**2.** El invariante del bucle del segundo ejemplo se dio en términos de un diagrama (véase (15.1.6)). Reemplace el diagrama por una instrucción equivalente en el cálculo de predicados.

3. Escriba un programa que, dada una matriz entera fija *b [0:n —1],* donde *n* >0, establezca x en el valor más pequeño de *b.*  El programa debe ser no determinista si el valor más pequeño ocurre más de una vez en *b .* La precondición *Q,* la postcondición *R, el* invariante de bucle *P* y la función enlazada *t* son

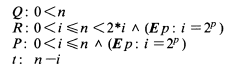
1. *0<n*
2. *x -... c,b[0:n* —1] A *(E j: 0 --j <n: x = KID*

*P: 1,-..* *i --.\_n* A *x --.\_b[0:i* —1] A *(Ej:0-.* *.. j<i: x=bUD t: n—i*

4. Escribe un programa para el problema del ejercicio 3, pero usa el invariante y el límite función

*P: 0<i <n* A x <*b[i:n-1]* A *(Ej: i< j<n: x =b[j]) t: i*

5. Escriba un programa que, dado un entero fijo *n* >0, establezca la variable *i* a la potencia más alta de 2 que es como máximo *n.*  La precondición *Q,* postcondición *R,* bucle invari­ant *P* y función enlazada *t* son

1. *0<n*
2. *0<i <n <2\*i* A *(Ep: i =2P)*

*P: 0<i <n* A *(Ep: i =2k)*

*t: n—i*

6. Traducir el programa (15.1.7) al idioma de su elección —PL/ I, Pascal, FORTRAN, etc.— recordando la necesidad de la operación **cand.**  Compare su respuesta con (15.1.7).

**15.2 Avanzar hacia la terminación**

***Ordenar cuatro tuplas***

Considere el siguiente problema. Escriba un programa que ordene las cuatro variables *enteras q0, q1, q2, q3.*  Es decir, al finalizar debe ser cierto lo siguiente: *q0 q1 q2 q3.*

Implícito está el hecho de que los valores de las variables deben ser permutados —por ejemplo, la asignación *q0, q1, q2, q3 := 0, 0, 0, 0* no es una solución, aunque establece *q0 q1 q2 q3.* Para transmitir esta información explícitamente, usamos **Qi**para denotar el valor inicial de **qi***,* y escribimos la­ especificación formal

*P:*

*Q:*

donde la segunda *conjunción perm (…, …)* de *Q* significa que las cuatro variables *q0, q1, q2, q3* contienen una permutación de sus valores originales.

Se escribirá un bucle. Su invariante expresa el hecho de que las cuatro variables deben contener siempre una permutación de sus valores iniciales:

I: perm ((q0, q1, q2, q3), (Q0, Q1, Q2, Q3))

La función variante es el número de *inversiones* en la secuencia *(q0, q1, q2, q3).* Para una secuencia, el número de inversiones es el número de pares*, i < j,* que están fuera de orden, es decir*, .*

Tenga en cuenta que esto **incluye *todos los* pares**, y no solo adyacentes. Por ejemplo, el número de inversiones en **(1,** 3, 2, 0) es 4. Así que La función variante es

*t: #{}.*

El invariante indica que las cuatro variables deben contener siempre una permutación de sus valores iniciales. Esto es obviamente cierto inicialmente, por lo que no se necesita inicialización.

**En la última sección**, en este punto del desarrollo **se determinó la guardia del bucle**. En cambio, **aquí buscaremos una serie de comandos protegidos, cada uno de los cuales avanza hacia la terminación**. El invariante indica que los únicos comandos posibles son aquellos que intercambian (permutan) los valores de dos o más de las variables. Para mantener las cosas­ simples, considere solo los intercambios de dos variables. Hay seis posibilidades: *q0, q1 := q1, q0* y *q1, q2:= q2, q1,* etc.

Ahora, la ejecución de un comando debe avanzar hacia la ­terminación. Considere un posible comando, *q0, q1 := q1, q0.*  Disminuye el número de inversiones en *(q0,q1,q2, q3) q0 > q1.* Por lo tanto, el comando protegido servirá. Cada una de las otras 5 posibilidades ­son similares, y juntas producen el programa.

**do**

**|**

**|**

**|**

**|**

**|**

**od**

Todavía queda por demostrar que al finalizar se establece el resultado *Q* —este es el punto 3 de la lista de verificación **11.9,**  ***.*** Supongamos que todos los guardias son falsos. Entonces (porque el primer guardia es falso), (porque el segundo es falso) y (porque el tercero es falso); por lo tanto,

*.*

Junto con el invariante ***I,*** esto implica el resultado deseado. Pero tenga en cuenta que solo se necesitaron los primeros tres guardias para establecer el resultado deseado. Por lo tanto, los últimos tres comandos protegidos se pueden eliminar, lo que produce el programa

**do**

**|**

**|**

**od**

***Discusión***

El enfoque utilizado aquí se puede resumir de la siguiente manera.

(15.2.1) **•Estrategia para desarrollar un bucle:** Desarrollar comandos protegidos, creando cada comando para que avance hacia la terminación y creando la correspondiente guarda ­para garantizar que el invariante se­ mantenga. El proceso de desarrollo de comandos protegidos finaliza cuando se han desarrollado suficientes de ellos para probar ***.***

Desarrollar los comandos como se indica garantiza que los puntos 2, **4** y 5 de la lista de verificación 11.9 sean ciertos. La última oración de la estrategia indica que el bucle se completa cuando el punto 3 de la lista de verificación es verdadero. Por supuesto, puede ­ser necesario escribir inicialmente la inicialización para que el invariante sea verdadero (punto 1 de la lista de comprobación).

La estrategia hace hincapié en los puntos 2 y 4 de la lista de comprobación 11.9, que se refieren a los progresos realizados hacia la terminación y el mantenimiento de ­la invariancia. En el enfoque utilizado en la sección 15.1, se hizo hincapié en primer lugar en demostrar el punto 3, que al finalizar el resultado ***Q*** es verdadero.

Discutamos el paso aparentemente mágico de eliminar tres comandos guardados del bucle. Una vez que se ha desarrollado un bucle correcto, a veces se puede derivar uno más corto y quizás más eficiente. Cada comando protegido ya satisface los puntos 2 y **4** de la lista de verificación 11.9. El fortalecimiento de los guardias no puede destruir el hecho de que los puntos 2 y 4 están satisfechos, de modo que los guardias se pueden cambiar a voluntad, siempre que se fortalezcan. El único problema es asegurarse de que al finalizar el resultado aún se mantenga, es decir, sigue siendo verdadero.

Si es posible fortalecer una guardia a *F* (false) sin violar ***,*** entonces el comando correspondiente nunca se puede ejecutar, por lo que el comando guardado se puede eliminar. Esto es lo que sucedió en este ejemplo. Solo se necesitaron los tres primeros guardias para probar de modo que los tres últimos podrían reforzarse a *F* y luego eliminarse.

Volveremos sobre este punto en el capítulo 19 sobre la eficiencia.

Este pequeño programa no es determinista en la ejecución, porque dos, e incluso tres, guardias pueden ser verdaderos al mismo tiempo. Pero, para cualquier estado inicial hay exactamente un estado final, de modo que en términos del resultado el programa es determinista.

El número de iteraciones del bucle es igual al número de inver­siones, que es como máximo 6.

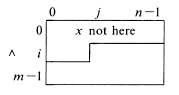
***Búsqueda de una matriz bidimensional***

Considere de nuevo un problema discutido en la sección 15.1:

Escribir un programa para buscar en una matriz bidimensional. La única diferencia en el problema es que aquí la matriz puede estar vacía (es decir, puede tener 0 filas o 0 columnas). La matriz fija es b [0:m-1, 0:n-1], donde y *,* y se debe buscar un entero fijo *x.*  Usando las variables *i* y *j,* al terminar­ *x = b[i,j]* o, si esto no es posible, *i = m.*  Para ser más ­preciso, Q debe establecerse:

(15.2.2) *Q: .*

El invariante *I,* dado a continuación usando un diagrama, establece que x no está en las filas ya buscadas b [0:i-1] y no en las columnas ya buscadas b [i, 0:j-1]de la fila actual *i.*

n

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | *x* no aquí | |
| (15.1.6) *I:* |  |  |  |
|  |  |  |  |
| *M-* |  |  |  |

La función variante es la suma del número de valores en la sección no probada y el número de filas en la sección no probada: *t = (m-i)\*n-j +m-i.*  El valor adicional *m - i* es necesario porque posiblemente *j* = *n.*  Como primer paso en el desarrollo, determine la inicialización del bucle.

La elección obvia es *i, j :=* 0, 0, porque entonces la sección en la que "x no está aquí" está vacía. Observe cuidadosamente cómo el invariante incluye *,* en lugar de *j < n.*  Esto es necesario porque el número de columnas, *n,* podría ser 0.

A continuación, se deben desarrollar comandos protegidos para el bucle. ¿Cuál es el comando *más simple* posible y cuál es un guardia adecuado para ello?

El comando obvio para probar es *j:=*j+1, porque disminuye *t.* (Otra posibilidad, que se investigará posteriormente, es *i:=*i+1). Un guardian adecuado debe garantizar que *I* sigue siendo cierto. Formal o informalmente, podemos ver que

Tenga en cuenta que esta guardia se ha hecho lo más débil posible. Ahora, ¿un bucle con este comando de protección única resuelve el problema? ¿Por qué o por qué no? Si no es así, ¿qué otro comando protegido se puede usar?

Un bucle con solo este comando guardado podría terminar con*,* y esto, junto con el invariante, no es suficiente para probar *Q.*  De hecho, si la primera fila de *b* no contiene x, **¡el bucle terminará después de buscar solo en la primera fila!** **Algún comando vigilado debe lidiar con el aumento *de i.***

El comando *i:=i+1* sólo puede ejecutarse si *i* < m. Además, tiene la posibilidad de mantener *I* true solo si la fila *i* no contiene x, por lo que con­siderar ejecutándola solo bajo la condición adicional *j = n.* Pero esto significa que *j* también debe establecerse en 0, de modo que se mantenga la condición en la fila actual *i*. Esto lleva al programa

(15.2.4) *i, j :=* 0, 0

**do**

**|**

**od**

Todavía queda demostrar que al finalizar *Q* es verdadero, es decir, *.* Supongamos que los guardias son falsos. Surgen dos casos. Primero, *i* = m podría sostenerse. En segundo lugar, si *,* entonces la falsedad de la segunda guardia implica *;*  por lo tanto, la falsedad de la primera guardia implica **x = b[i, j]***.*  Por lo que, si los guardianes son falsos, lo siguiente debe ser verdadero:

*,*

y esto junto con *I* implica el resultado *Q.* Por lo tanto, el programa es correcto. Nótese que en el caso *i = m* el invariante implica que x no está en las filas 0 a *m* —1 de *b,* lo que significa que x no está en b*.*

***Discusión***

Este bucle se desarrolló continuando desarrollando comandos simples y protegidos que progresaron hacia la terminación hasta ***.***  Esto llevó a un bucle con una forma radicalmente diferente de lo que la mayoría de los programadores ­están acostumbrados a desarrollar (**en parte porque por lo general no saben sobre comandos protegidos**). Lleva tiempo acostumbrarse (15.2.4) como un bucle para buscar una matriz bidimensional.

Este problema se utiliza a menudo para argumentar a favor de la inclusión de **gotos** o "exist" de bucle en un lenguaje convencional, porque, a menos que se use un variable adicional ­comúnmente llamado "fllag", la solución convencional al problema necesita dos bucles anidados y un "exist" del interno:

(15.2.5) *i, j:=* 0, 0;

**while do**

**begin while** **do**

**if** x = *b [i ,j]* **then goto** *loopexit*

***else*** *j := j +1;*

*i, j:=* i+1, 0

**end;**

*loopexit:*

Vemos, entonces, que la notación de comando protegida y el método de desarrollo juntos conducen a una solución más simple y fácil de entender al problema, siempre que se entienda la metodología.

¿Cómo podría ejecutarse eficazmente el programa (15.2.4)? Un compilador optimizador podría analizar las guardias y los comandos y determinar las rutas de ejecución dadas en el diagrama (15.2.6): en el diagrama, una flecha con *F (T)* representa la ruta que se debe tomar cuando el término del que emana es falso (verdadero). ¡Pero (15.2.6) es esencialmente un gráfico de defectos para el programa (15.2.5)! Por lo tanto, al menos en este caso, el programa de "alto nivel" ­(15.2.4) puede simularse utilizando los constructos de "nivel inferior" de Pas­cal, FORTRAN y **PL/I.**

El programa (15.2.4) se desarrolla a partir de principios sólidos. El programa (15.2.5) se desarrolla normalmente de manera *ad hoc*, utilizando el desarrollo por casos de prueba, el resultado es que se plantea la duda de si todos los casos han sido cubiertos.

