Diseño de algoritmos iterativos

Ramón González del Campo es el autor principal de este tema

Facultad de Informática - UCM

13 de octubre de 2013

Verificar es demostrar que las instrucciones de un algoritmo son correctas, es decir, que para una entrada válida (precondición) producen el resultado esperado (postcondición).

Ejemplo: Problema: Intercambiar el valor de dos variables

Especificación:

x,y : entero
$P : \{x = X \wedge y = Y\}$
Intercambiar
$Q : \{x = Y \wedge y = X\}$

¿ Cuales de los siguientes algoritmos son correctos?

aux = x;
x = y;
y = aux;
	x = y;
	y = x;
		x = x - y;
		y = x + y;
		x = y - x;
			x = x + y;
			y = x - y;
			x = y - x;

Verificación

- Para verificar un algoritmo se definen predicados intermedios entre cada instrucción elemental, llamados *aserciones* o *asertos*:

$$\{R_0\}A_1\{R_1\}; \ldots; \{R_{n-1}\}A_n\{R_n\}$$

- Si para cada instrucción A_k se satisface $\{R_{k-1}\}A_k\{R_k\}$ y $P \Rightarrow R_0$ y $R_n \Rightarrow Q$ entonces se satisface $\{P\}A\{Q\}$.
- La semántica del lenguaje define para cada tipo de instrucción del lenguaje las reglas que determinan si se satisface $\{R_{k-1}\}A_k\{R_k\}$ (reglas de verificación).

Verificación

Axioma de asignación Para toda variable x, toda expresión válida del mismo tipo E y todo predicado R, la especificación:

$$\boxed{\begin{array}{l} P : \{Dom(E) \wedge \{R_x^E\} \\ \qquad \mathrm{x} = \mathrm{E} \\ Q : \{R\} \end{array}}$$

es correcta.

P se denomina precondición más débil (*pmd*).

- $Dom(E)$ el conjunto de todos los estados en los que la expresión E está definida.
- R_x^E el predicado resultante de sustituir toda aparición de x por E en el predicado R.

Ejemplo: Suponiendo x entero determina la precondición más débil,P, que safisfaga la especificación:

P :	$Q_x^{x+2} : \{x + 2 \geq 0\} \equiv \{x \geq -2\}$
$\mathrm{x} = \mathrm{x} + 2$	$\mathrm{x} = \mathrm{x} + 2 \quad \uparrow$
$Q : \{x \geq 0\}$	$Q : \{x \geq 0\}$

Regla de inferencia de la asignación

La especificación $\{P\}\ \mathrm{x = E}\ \{Q\}$ es correcta si $P \Rightarrow Q_x^E$.

Regla de inferencia de la composición secuencial

La especificación $\{P\}\ A_1; A_2\ \{Q\}$ es correcta si existe un predicado R tal que las especificaciones $\{P\}\ A_1\ \{R\}$ y $\{R\}\ A_2\ \{Q\}$ son correctas.

Ejemplo: Suponiendo x, y enteros, calcula el predicado P más débil que satisfaga la especificación:

P :	$R_2 \equiv R_{1_x}^{x*x} : \{x * x + 1 > 0\} \equiv \{true\}$
$\mathrm{x = x * x}$	$\mathrm{x = x * x}$ $\uparrow$
	$R_1 \equiv Q_x^{x+1} : \{x + 1 > 0\}$
$\mathrm{x = x + 1}$	$\mathrm{x = x + 1}$ $\uparrow$
$Q : \{x > 0\}$	$Q : \{x > 0\}$

Regla de inferencia de la composición alternativa (I)

La especificación

$$\boxed{\begin{array}{l}\{P\}\\ \quad \texttt{if}\ (B)\ A_1\ \texttt{else}\ A_2;\\ \{Q\}\end{array}}$$

es correcta si las especificaciones

$$\boxed{\begin{array}{l}\{P \wedge B\}\\ \quad A_1\\ \{Q\}\end{array}} \quad \text{y} \quad \boxed{\begin{array}{l}\{P \wedge \neg B\}\\ \quad A_2\\ \{Q\}\end{array}}$$

son correctas. La *pmd* es $B \wedge pmd(A_1, Q)) \vee (\neg B \wedge pmd(A_2, Q))$

Ejemplo:

P :	$P \wedge x \geq 0 \Rightarrow R_1$	$P \wedge \neg(x \geq 0) \Rightarrow R_2$
if (x <= 0) y = x;	$R_1 \equiv Q_y^x : \{x = Y\}$	$R_2 \equiv Q_y^{-x} : \{-x = Y\}$
else y = -x;	y = x $\uparrow$	y = -x $\uparrow$
$Q : \{y = Y\}$	$Q : \{y = Y\}$	$Q : \{y = Y\}$

$pmd \equiv (x \geq 0 \wedge x = Y) \vee (\neg(x \geq 0) \wedge -x = Y)$

Regla de inferencia de la composición iterativa

La especificación $\boxed{\{P\}\ \texttt{while}\ (B)\ \texttt{do}\ A\ \{Q\}}$ es correcta si existe un predicado I que llamaremos invariante y una función t dependiente de las variables que intervienen en el proceso y que toma valores enteros, que llamaremos función cota, de forma que:

1. $P \Rightarrow I$ el invariante se cumple antes de entrar al bucle
2. $\{I \wedge B\}\ A\ \{I\}$ vuelve a iterar
3. $I \wedge \neg B \Rightarrow Q$ si no concuerda salta del bucle
4. $I \wedge B \Rightarrow t > 0$ si entra es porque la cota es mayor que cero
5. $\{I \wedge B \wedge t = T\}\ A\ \{t < T\}$ cuando da una vuelta la cota disminuye de la cota inicial

Existen muchos predicados invariantes, se ha de elegir uno que nos permita verificar las 5 condiciones de corrección del bucle, esto es

- Lo suficientemente fuerte para $I \wedge \neg B \Rightarrow Q$
- Lo suficientemente débil para $P \Rightarrow I$.

El invariante es un predicado que describe todos los estados por los que atraviesa el cómputo realizado por el bucle, observados justo antes de evaluar la condición B de terminación.

Ejemplo:

$i = 0;\ q = 0;\ p = 1;$
$P : \{i = 0 \wedge q = 0 \wedge p = 1\}$
while $(i < n)\{$
    $q = q + p;\ p = p + 2;\ i = i + 1;$
$\}$
$Q : \{i = n \wedge q =? \wedge p =?\}$

valores de las variables antes de eval. cond. $i < n$.

estado	i	q	p
P_0	0	0	1
P_1	1	1	3
P_2	2	4	5
P_3	3	9	7
...			

¿ Que relaciones entre las variables se mantienen durante la ejecución del bucle?

Un invariante del bucle es: $\boxed{I : \{0 \leq i \leq n \wedge q = i^2 \wedge p = 2i + 1\}}$

Verificación

La *función cota* o *función limitadora* es una función $t : estado \to \mathcal{Z}$ positiva que decrece cada vez que se ejecuta el cuerpo del bucle.

La función cota garantiza la terminación del bucle

Para encontrar una función cota se observan las variables que son modificadas por el cuerpo A del bucle, y se construye con ellas una expresión entera t que decrezca

Ejemplo: cálculo del cuadrado de un número.
Las variables i, p y q crecen, n se mantiene inalterable, por lo tanto $n - i$ decrece.
Además, la condición del bucle $i \leq n$ garantiza que la función es positiva.

La función cota es : $t = n - i$.

Ejemplos

Verifica el siguiente algoritmo respecto a su especificación.

$\{N \geq 0\}$
fun *suma*(**int** *V[N]*) **return int** x
$\{x = (\Sigma i : 0 \leq i < N : V[i])\}$

```
int suma(int V[], int N){
  int n;
  n = N;   x = 0;
  while (n != 0)
  {
    x = x+V[n-1];
    n = n-1;
  }
  return x;
}
```

Invariante: $I \equiv \{0 \leq n \leq N \wedge x = (\Sigma i : n \leq i < N : V[i])\}$.

Ejemplos

- Demostrar la corrección de la siguiente especificación suponiendo x, y, n: enteros.

  $\{n \geq 0\}$

  ```
  int x, y;
  x = 0;  y = 1;
  while (x != n)
  {
    x = x+1;
    y = y+y;
  }
  ```

  $\{y = 2^n\}$

- Invariante: $I \equiv \{0 \leq x \leq n \wedge y = 2^x\}$

Derivación

Derivar: construir las instrucciones de un algoritmo a partir de su especificación asegurando su corrección.

Los algoritmos se ajustan al esquema:

$\{P\}$
A_0 `(Inicialización)`
$\{I, Cota\}$
while `(B) {`
  $\{I \wedge B\}$
  A_1 `(Restablecer)`
  $\{R\}$
  A_2 `(Avanzar)`
  $\{I\}$
`}`
$\{Q\}$

- A_0 Hace que el invariante se cumpla inicialmente A_2 hace que la cota decrezca.
- A_1 mantiene el invariante a cierto.

- **Pasos** para construir un algoritmo con bucle:
  1. Diseñar el invariante y la condición del bucle sabiendo que se tiene que cumplir:
     $I \wedge \neg B \Rightarrow Q$
  2. Diseñar A_0 para hacer el invariante cierto: $\{P\}A_0\{I\}$
  3. Diseñar la función cota, C, de tal forma que: $I \wedge B \Rightarrow C \geq 0$.
  4. Diseñar A_2 y el predicado $R \equiv pmd(A_2, I)$.
  5. Diseñar A_1 para que se cumpla: $\{I \wedge B\}A_1\{R\}$.
  6. Comprobar que la cota realmente decrece:

$$\{I \wedge B \wedge C = T\}A_1, A_2\{C < T\}$$

- Deriva un algoritmo que verifique la siguiente especificación:

  $\{b \neq 0\}$
  proc *divide(***int** *a,b;* **out int** *c,r)*
  $\{a = b * c + r \wedge 0 \leq r < b\}$

- Deriva un algoritmo que verifique la siguiente especificación:

  $\{n \neq 0\}$
  fun *raiz-entera(***int** *n;)* **return int** *r*
  $\{r \geq 0 \wedge r^2 \leq n < (r+1)^2\}$

Reglas prácticas para el cálculo de la eficiencia

- Las instrucciones de asignación, de entrada-salida, los accesos a elementos de un vector y las expresiones aritméticas y lógicas, (siempre que no involucren variables cuyos tamaños dependan del tamaño del problema) tendrán un coste constante, $\Theta(1)$. No se cuentan los *return*.
- Para calcular el coste de una composición secuencial de instrucciones, $S_1; S_2$ se suma los costes de S_1 y S_2. Si el coste de S_1 está en $\Theta(f_1(n))$ y el de S_2 está en $\Theta(f_2(n))$, entonces: $\Theta(f_1(n)) + \Theta(f_2(n)) = \Theta(max(f_1(n), f_2(n)))$.

- Para calcular el coste de una instrucción condicional:

$$\textbf{if } (B)\ \{S_1\}\ \textbf{else}\ \{S_2\}$$

  Si el coste de S_1 está en $\mathcal{O}(f_1(n))$, el de S_2 está en $\mathcal{O}(f_2(n))$ y el de B en $\mathcal{O}(f_B(n))$, podemos señalar dos casos para el condicional:

  - *Caso peor*: $\mathcal{O}(max(f_B(n), f_1(n), f_2(n))$.
  - *Caso promedio*: $\mathcal{O}(max(f_B(n), f(n))$ donde $f(n)$ es el promedio de $f_1(n)$ y $f_2(n)$.

  Se pueden encontrar expresiones análogas a éstas para la clase omega.

- Bucles con la forma:
  while (B) $\{S\}$
  Para calcular el coste de tal bucle hay que calcular primero el coste de cada pasada y después sumar los costes de todas las pasadas que se hagan en el bucle. El número de iteraciones dependerá de lo que tarde en hacerse falso B, teniendo en cuenta los datos concretos sobre los que se ejecute el programa y los grandes que sean.
- De manera práctica, se analizará el código de los algoritmos presentados de arriba hacia abajo y de dentro hacia fuera.

- Búsqueda secuencial

```
bool buscaSec( int v[], int num, int x ) {
  int j;
  bool encontrado;

  j = 0;
  encontrado = false;
  while ( (j < num) && ! encontrado ) {
    encontrado = ( v[j] == x );
    j++;
  }
  return encontrado;
}
```

- Búsqueda binaria

```
int buscaBin( int v[], int num, int x ) {
  int izq, der, centro;

  izq = -1;
  der = num;
  while ( der != izq+1 ) {
    centro = (izq+der) / 2;
    if ( v[centro] <= x )
    izq = centro;
    else
    der = centro;
  }
  return izq;
}
```

- Ordenación por inserción

```
void ordenaIns ( int v[], int num ) {
  int i, j, x;

  for ( i = 1; i < num; i++ ) {
    x = v[i];
    j = i-1;
    while ( (j >= 0 ) && (v[j] > x) ){
      v[j+1] = v[j];
      j = j-1;
    }
    v[j+1] = x;
  }
}
```