Resumen Final Algo 1

Especificación de problemas

¿Qué es especificar un problema?

Dado un problema, especificarlo es dar una **descripción** de lo que se debe resolver. Es un **contrato** que define **qué se debe resolver** y **qué propiedades debe tener la solución**.

¿Para qué sirve la especificación?

- Testing
- **Demostración formal de correctitud**

- Construir un programa a partir de su especificación

¿Qué es un contrato?

El contrato es la **garantía** de que un programa cumple con su especificación.

El programador escribe un programa P tal que **si el usuario suministra datos que hacen verdadera la precondición**, entonces **P termina** en una cantidad finita de pasos **retornando un valor que hace verdadera la postcondición**.

¿Cuándo un programa es correcto?

El programa P es **correcto** respecto de su especificación exactamente cuando se cumple el contrato.

Sobre-especificación

Consiste en dar una **postcondición más restrictiva** que lo que se necesita, o bien dar una **precondición más débil** que la que se podría dar.

- **Postcondición más restrictiva:** limita los posibles algoritmos que resuelven el supuesto problema
- **Precondición más débil:** amplía los datos de entrada permitiendo datos inválidos
- **Sobre-especifico, le sobran propiedades a mi postcondición**

Sub-especificación

Consiste en dar una **precondición más restrictiva** que lo que se necesita, o bien dar una **postcondición más débil** que la que se podría dar.

- **Precondición más restrictiva:** deja afuera datos de entrada perfectamente válidos
- **Postcondición más débil:** ignora condiciones necesarias para la salida (permite soluciones no deseadas)
- **Sub-especifico, le faltan propiedades a mi postcondición**

Relación de fuerza

A es más fuerte que B si $(A \rightarrow B)$ es una tautología.

Correctitud y Teorema del Invariante

Estado

Un estado de un programa es el conjunto de valores de todas las variables de dicho programa en un punto de su ejecución.

La asignación es la instrucción que permite pasar de un estado al siguiente.

Tripla de Hoare

Cuando un programa S es correcto respecto de la especificación (P, Q), lo denotamos con la **tripla de Hoare {P} S {Q}**.

Invariante de un ciclo

Un predicado I es un invariante de un ciclo si:

1. I vale antes de comenzar el ciclo, y
2. si vale $I \wedge B$ al comenzar una iteración arbitraria, entonces sigue valiendo I al finalizar la ejecución del cuerpo del ciclo

Un invariante describe el estado que se satisface cada vez que comienza la ejecución del cuerpo de un ciclo, y también se cumple cuando la ejecución del cuerpo del ciclo concluye.

Un buen invariante debe incluir:

- El rango de las variables de control del ciclo ($0 \leq i \leq n$)
- Alguna información sobre el acumulador del ciclo ($s = \sum_{k=1}^{i-1} k$)

El invariante del ciclo caracteriza las acciones del ciclo, y representa al algoritmo que el ciclo implementa.

Teorema del invariante

Si existe un predicado I tal que:

1. $PC \Rightarrow I$, **(I se cumple al principio del ciclo)**
2. $\{I \wedge B\}\ S\ \{I\}$, **(I se preserva en cada iteración)**
3. $I \wedge \neg B \Rightarrow QC$, **(Se cumple QC a la salida del ciclo)**

entonces el ciclo while (B) S es **parcialmente** correcto respecto de la especificación (PC , QC).

Precondición más débil

Precondición más débil

La precondición más débil de un programa S respecto de una postcondición Q es el predicado P más débil posible tal que {P} S {Q}.

Se denota wp(S, Q).

Teorema

Una tripla de Hoare {P} S {Q} es válida $\Leftrightarrow$ $(P \Rightarrow_L wp(S, Q))$

Axiomas de wp

1. **Asignación:** $wp(x := E, Q) = def(E) \wedge_L Q\{x \leftarrow E\}$
2. **skip:** $wp(skip, Q) = Q$
3. **Secuencia:** $wp(S1; S2, Q) = wp(S1, wp(S2, Q))$
4. **Condicional:** $wp(\text{if B then S1 else S2 endif}, Q) = def(B) \wedge_L ((B \wedge wp(S1, Q)) \vee (\neg B \wedge wp(S2, Q)))$
5. **Ciclo:** $wp(\text{while B do S endwhile}, Q) = (\exists\ i \geq 0)\ (Hi(Q))$, donde Hk(Q) es el conjunto de estados a partir de los cuales la ejecución del ciclo termina en exactamente k iteraciones.

Propiedades de wp

1. **Monotonía:** Si $Q \Rightarrow R$ entonces $wp(S, Q) \Rightarrow wp(S, R)$
2. **Distributividad:** $wp(S, Q) \wedge wp(S, R) \Rightarrow wp(S, Q \wedge R)$
3. **“Excluded miracle”:** $wp(S, false) = false$
4. **Corolario monotonía:** Si $P \Rightarrow wp(S1, Q)$ y $Q \Rightarrow wp(S2, R)$ entonces $P \Rightarrow wp(S1; S2, R)$

Precondición más débil de ciclos

Teorema de terminación

Si existe una función $fv : V \rightarrow Z$ tal que

1. $\{I \wedge B \wedge v0 = fv\}\ S\ \{fv < v0\}$, y **(La función variante es estrictamente decreciente)**

2. $I \wedge fv \leq 0 \Rightarrow \neg B$, **(si la función variante alcanza la cota inferior la guarda se deja de cumplir)**

entonces la ejecución del ciclo while (B) S **siempre termina**.

La función fv se llama **función variante** del ciclo.

Una función variante representa una cantidad que se va reduciendo a lo largo de las iteraciones.

Teorema de corrección de un ciclo (Teorema del Invariante + Teorema de terminación)

Sean un predicado I y una función $fv : V \rightarrow Z$ (donde V es el producto cartesiano de los dominios de las variables del programa). Si

1. $PC \Rightarrow I$, **(I se cumple al principio del ciclo)**
2. $\{I \wedge B\}\ S\ \{I\}$, **(I se preserva en cada iteración)**
3. $I \wedge \neg B \Rightarrow QC$, **(Se cumple QC a la salida del ciclo)**
4. $\{I \wedge B \wedge v0 = fv\ \}\ S\ \{fv < v0\ \}$, **(La función variante es estrictamente decreciente)**
5. $I \wedge fv \leq 0 \Rightarrow \neg B$, **(Si la función variante alcanza la cota inferior la guarda se deja de cumplir)**

entonces la siguiente tripla de Hoare es válida: { PC } while B do S endwhile { QC }

Corrección de programas en C++

1. Traducir el programa C++ a SmallLang preservando su comportamiento.
2. Demostrar la corrección del programa en SmallLang con respecto a la especificación.
3. Entonces, probamos la corrección del comportamiento del programa original.

Testing

¿Qué es hacer testing?

Es el proceso de **ejecutar** un producto para:

- Verificar que satisface los requerimientos (en nuestro caso, la especificación).
- Identificar diferencias entre el comportamiento real y el comportamiento esperado

Objetivo: encontrar defectos en el software

Definiciones

- **Programa bajo test:** Es el programa que queremos testear
- **Test input (o dato de prueba):** Es una asignación concreta de valores a los parámetros de entrada para ejecutar el programa bajo test.
- **Test case:** Es un código que ejecuta el programa a testear usando un test input, y chequea automáticamente si se cumple la condición de aceptación sobre la salida.
- **Test suite:** Es un conjunto de test cases.

Limitaciones del testing

Una de las mayores dificultades es encontrar un conjunto de tests adecuados:

- Suficientemente grande para abarcar el dominio y maximizar la probabilidad de encontrar errores.
- Suficientemente pequeño para poder ejecutar el proceso con cada elemento del conjunto y minimizar el costo del testing (es decir, que sea viable)

Al no ser exhaustivo, el testing NO puede demostrar que el software funciona correctamente.

Control-Flow-Graph (CFG)

Es una representación gráfica del programa.

Criterios de adecuación estructurales

- **Sentencias:** cubrir todos los nodos del CFG.
- **Arcos:** cubrir todos los arcos del CFG.
- **Decisiones (branches):** por cada if, while, for, etc., la guarda fue evaluada a verdadero y a falso.
- **Condiciones básicas:** por cada componente básica de una guarda, este fue evaluado a verdadero y a falso.
- **Caminos:** cubrir todos los caminos del CFG. Como no está acotado o es muy grande, se usa muy poco en la práctica.

Buenas prácticas de programación

Nombres declarativos

Usar nombres que revelen la intención de los elementos nombrados. El nombre de una variable/función debería decir todo lo que hay que saber sobre ella.

Se debe tener un nombre por concepto. No tener funciones llamadas grabar, guardar y registrar.

Comentarios

- Explican la intención del programador
- Explicitan precondiciones o suposiciones
- Clarifican código que a primera vista puede no ser claro

Variables

En C/C++ podemos tener variables declaradas pero no inicializadas, y el valor que contienen en ese caso es impredecible (decimos que contienen “basura”).

Para evitar esta situación, es recomendable **inicializar siempre** las variables.

Usar el scope más pequeño posible

Funciones

Cada función debe:

1. Hacer **una sóla cosa**,
2. Hacerla **bien**, y
3. Ser el **único** componente del programa encargado de esa tarea.

Búsqueda sobre secuencias

Búsqueda lineal

Problema: Buscar un elemento en una secuencia.

Especificación:

```
proc contiene(in s : seq<Z>, in x : Z, out result : Bool) {
  Pre { True }
```

Post { result = true ⇔ (∃ i : Z) (0 ≤ i < |s| ∧L s[i] = x) }

}

Idea: Recorremos la secuencia linealmente y, por cada elemento, nos fijamos si es el que buscamos.

Invariante de ciclo: I ≡ 0 ≤ i ≤ |s| ∧L (∀ j : Z) (0 ≤ j < i ⇒L s[j] ≠ x)

Función variante: fv = |s| - i

Complejidad: O(n), con n la longitud de la secuencia.

Búsqueda binaria

Problema: Buscar un elemento en una secuencia **ordenada**.

Especificación:

proc contieneOrdenada(in s : seq<Z>, in x : Z, out result : Bool) {

Pre { ordenado(s) }

Post { result = true ⇔ (∃ i : Z) (0 ≤ i < |s| ∧L s[i] = x) }

}

Idea: Comparamos el valor a buscar con el elemento del medio de la secuencia. Si son iguales, ya encontré el elemento. Si no son iguales, vemos si es menor que el elemento del medio. Si es menor, descarto la segunda mitad de la secuencia (ya que todos esos elementos son mayores que el elemento a buscar) y repito la búsqueda con la primer mitad de la secuencia. Si no es menor, entonces es mayor, y descarto la primer mitad de la secuencia (ya que todos esos elementos son menores que el elemento a buscar) y repito la búsqueda con la segunda mitad de la secuencia.

Invariante de ciclo: I ≡ 0 ≤ low < high < |s| ∧L s[low] ≤ x < s[high]

Función variante: fv = high - low - 1

Complejidad: O(log n), con n la longitud de la secuencia.

Ordenamiento sobre secuencias

Selection Sort

Problema: Ordenar una secuencia.

Especificación:

proc ordenar(inout s : seq<Z>) {

Pre { s = S0 }

Post { mismos(s, S0) ∧ ordenado(S0) }

}

Idea: Dada una posición i, busco el elemento que tiene que ir en dicha posición. Recorremos la secuencia linealmente y, por cada posición i, buscamos el mínimo elemento entre las posiciones i y |s| -1. Luego lo intercambiamos por el iésimo elemento.

Invariante de ciclo: I ≡ mismos(s, S0) ∧ (0 ≤ i ≤ |s|) ∧L ordenado(s, 0, i) ∧ (∀ j, k : Z) ((0 ≤ j < i ∧ i ≤ k < |s|) ⇒L s[j] ≤ s[k])

Función variante: fv = |s| - i

Complejidad: O(n2), con n la longitud de la secuencia.

Insertion Sort

Problema: Ordenar una secuencia.

Especificación:

proc ordenar(inout s : seq<Z>) {

Pre { s = S0 }

  Post { mismos(s, S0) ∧ ordenado(S0) }

}

Idea: Dado un elemento x, voy llevando a x a la posición en la que debe estar. Recorremos la secuencia linealmente y, por cada elemento, vamos intercambiandolo por los elementos a su izquierda, uno por uno, siempre y cuando estos sean mayores. Cuando encuentro un elemento menor, termino y avanzo con el siguiente elemento.

Invariante de ciclo: I ≡ mismos(s, S0) ∧ (0 ≤ i ≤ |s|) ∧L ordenado(s, 0, i)

Función variante: fv = |s| - i

Complejidad: O(n2), con n la longitud de la secuencia.

Dutch National Flag Problem

Problema: Dada una secuencia que contiene colores (rojo, blanco y azul), ordenarlos de modo que respeten el orden de la bandera holandesa (primero rojo, luego blanco y luego azul).

Especificación:

proc dutchNationalFlag (inout s : seq<Z>) {

  Pre { s = S 0 ∧ (∀e : Z) (e ∈ s ⇔ (e = 0 ∨ e = 1 ∨ e = 2)) }

  Post { mismos(s, S 0) ∧ ordenado(s) }

}

Idea: En una pasada, cuento la cantidad de apariciones de cada color, almacenándolas en tres variables. En una segunda pasada, relleno la secuencia, primero con la cantidad de rojos, luego con la cantidad de blancos y por último con la cantidad de azules.

Complejidad: O(2×n) = O(n), con n la longitud de la secuencia.

Otros algoritmos

- **Quicksort:** O(n2)
- **BubbleSort:** O(n2)
- **Mergesort:** O(n × log n)
- **Heapsort:** O(n × log n)

Algoritmos sobre secuencias ya ordenadas

Merge

Problema: Dadas dos secuencias ordenadas, unir ambas secuencias en una única secuencia ordenada.

Especificación:

proc merge(in a, b : seq<Z>, out result : seq<Z>) {

  Pre { ordenado(a) ∧ ordenado(b) }

  Post { ordenado(result) ∧ mismos(result, a ++ b) }

}

Idea: Tomo dos índices i y j, que corresponden a la secuencia a y b respectivamente. Comparo a[i] con b[j] y pongo el menor de ellos en un arreglo nuevo. Avanzo i o j según cuál haya elegido y repito.

Invariante de ciclo:

$$I \equiv ordenado(a) \ \wedge ordenado(b) \ \wedge |c| = |a| + |b|$$
$$\wedge \left(\left(0 \leq i \leq |a| \wedge 0 \leq j \leq |b| \wedge k = i + j\right) \wedge_L \left(mismos\Big(subseq(a, 0, i) \ ++ subseq(b, 0, j), \ subseq(c, 0, k \right.\right.$$
$$\wedge \ ordenado\Big(subseq(c, 0, k)\Big)\Big)\Big)$$
$$\wedge (i < |a|) \Rightarrow_L (\forall \, t : Z)\big(0 \leq t < j \Rightarrow_L b[t] \leq a[i]\big)$$
$$\wedge (j < |b|) \Rightarrow_L (\forall \, t : Z)\big(0 \leq t < i \Rightarrow_L a[t] \leq b[j]\big)$$

Función variante: fv = |a| + |b| - k

Complejidad: O(|a| + |b| + |c|)

String matching

Búsqueda de un patrón en un texto

Problema: Buscar un patrón en un texto.

Especificación:

```
proc contiene(in t, p : seq<Char>, out result : Bool) {
  Pre { True }
  Post { result = true ⇔ (∃ i : Z) (0 ≤ i ≤ |t| - |p| ∧L subseq(t, i, i + |p|) = p) }
}
```

```
proc contiene(in t, p : seq<Char>, out result : Bool) {
Pre { True }
Post { result = true ⇔ (∃ i : Z) (0 ≤ i ≤ |t| - |p| ∧L subseq(t, i, i + |p|) = p) }
}
```

Idea: Recorrer todas las posiciones i del texto t y, para cada una, verificar si subseq(t, i, t + |p|) = p

Complejidad: O(|t| * |p|)

Knuth, Morris y Pratt (KMP)

Problema: Buscar un patrón en un texto.

Especificación:

```
proc contiene(in t, p : seq<Char>, out result : Bool) {
  Pre { True }
  Post { result = true ⇔ (∃ i : Z) (0 ≤ i ≤ |t| - |p| ∧L subseq(t, i, i + |p|) = p) }
}
```

Idea: Tratar de no reanalizar **todo** el patrón cada vez que avanzamos en el texto.

Complejidad: O(|t| + |p|)