Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2012

Departamento de Computación - FCEyN - UBA

Programación imperativa - clase 2

Ciclos, invariantes, terminación

Semántica de programas imperativos

Tres instrucciones:

- asignación (y llamadas a funciones),
- condicional, y
- ciclo

Un programa es una lista de instrucciones, que se ejecutan una tras otra.

Semántica del programa se define mediante la composición de la semántica de las instrucciones por transformación de estados.

Asignación

Supongamos que en un programa dado, x es una variable, y $z_1, \ldots, z_k$ todas las variables distintas a x.

Si e es una expresión cuya evaluación no modifica el estado.

```
  //estado a
x = e;
  //estado b
  //vale x == e@a ∧ z1 = z1@a ∧ ··· ∧ zk = zk@a
```

Si e es la invocación a una función `f` que recibe parámetros por referencia, puede haber más cambios que se obtienen de la postcondifción de f .

Condicional

`if (B) {bloque uno;} else {bloque dos;}`

B es una expresión booleana del lenguaje de programación, sin efectos secundarios (no tiene que modificar el estado). Se llama guarda. `bloque uno` y `bloque dos` son bloques de instrucciones.

La semántica es por transformación de estados del condicional es:

```
  //estado a
if (B)  {
      //vale B@a
    bloque uno;
      //vale G
}
else {
      //vale not(B@a)
    bloque dos;
      //vale H
}
  //estado b
  //vale ( G ∨ H)
```

donde G y H son predicados que describen respecivamente los estados después de la ejecución de `bloque uno` y `bloque dos`.

Correctitud de un condicional

Especificamos un condicional con
una pre condición P_{if} , y
una post condición Q_{if} .

El condicional es correcto respecto de su especificación si y solamente si:
el estado previo al condicional fuerza la verdad P_{if}, y
el estado inmediatamente posterior al condicional cumple Q_{if}.

Para demostrar la correctitud de un condicional realizamos la transformación de estados, y demostramos las sentencias implica.

```
    //estado a
    //implica P_if
if (B)
    bloque uno;
else
    bloque dos;
  //estado b
  //implica Q_if
```


Ejemplo de demostración de condicional

problema $max(x, y : \text{Int}) = result : \text{Int}\{$
asegura $Q : (x > y \wedge result == x) \vee (x \leq y \wedge result == y)$
$\}$

Sea P_{if}: True
$Q_{if} : (x > y \wedge result == x) \vee (x \leq y \wedge result == y)$
B:$(x > y)$

```
int max(int x, int y) {
  int m;
    //estado antesdelif
    //implica P_if
  if (x > y) {
        m = x;
      //vale x > y ∧ m == x; }
  else {
        m = y;
      //vale x ≤ y ∧ m == y;
  }
    //estado despuesdelif
    //implica Q_if :  (x > y ∧ m == x) ∨ (x ≤ y ∧ m == y);
  return m;
  //implica Q;
}
```


Ciclos

```
while (B) cuerpo;
```

`B`: expresión booleana, sin efectos colaterales.
También se la llama guarda.

`cuerpo` es un bloque de instrucciones (entre llaves).
Se repite mientras valga la guarda `B`, cero o más veces.

La ejecución del ciclo termina si y solamente si el ciclo se repite una cantidad finita de veces. Y esto ocurre si y solamente si la guarda `B` llegar a ser falsa.

Si el ciclo termina, el estado resultante es el estado posterior a la última instrucción del cuerpo del ciclo.

Ejemplo de un ciclo

problema $fact(x : \text{Int}) = r : \text{Int}\{$
requiere $x \geq 0$
asegura $r == \prod[1..x]$
$\}$

En funcional:

```
fact :: Int -> Int
fact 0 = 1
fact i = i * (fact (i-1))
```

En imperativo:

```
int fact (int x) {
  int f = 1; int i = 0;
  while (i < x) {
    i = i + 1;
    f = f * i;
  }
  return f;
}
```

Correctitud de un ciclo

P_C: precondición del ciclo
Q_C: postcondición del ciclo
B: guarda
I: invariante

Sea B la expresión en el lenguaje de programación correspondiente a la guarda B.

```
int g(int x) {
  ....
  // estado antesdelciclo
  // vale P_C
  while (B) {
    cuerpo del ciclo (una o más instrucciones);
  }
  // estado despuesdelciclo
  // vale Q_C
  ....
}
```


Correctitud de un ciclo

P_C: precondición del ciclo,
Q_C: postcondición del ciclo,
B: guarda
I: invariante

```
int g (int x) {
  ....
  // estado antesdelciclo
  // vale P_C
  // implica I
  while (B) {
    // estado antes
    // vale B ∧ I
    cuerpo del ciclo (una o más instrucciones;
    // estado despues
    // vale I
  }
  // vale Q_C
  ....
}
```


Invariante

Expresión booleana del lenguaje de especificación que se mantiene verdadera, el los estados antes, durante y después de la ejecución de un ciclo.

- vale antes de entrar al ciclo (justo antes de evaluar la guarda)
- vale en cada iteración:

  justo después de entrar al cuerpo del ciclo, y

  justo después de ejecutar la última instrucción del cuerpo del ciclo,

  pero no vale en el medio del cuerpo.
- al salir del ciclo.
- es análogo a la hipótesis inductiva que usamos para demostrar la correctitud de un programa recursivo en programación funcional.
- conviene darlo al escribir el ciclo porque expresa la idea del ciclo.
- no hay forma algorítmica de encontrarlo

Un ejemplo

problema $sumat(x : \text{Int}) = r : \text{Int}\{$
  requiere $x \geq 0$
  asegura $r == \sum[0..x]$ }

En funcional:

```
sumat :: Int -> Int
sumat 0 = 0
sumat i = i + (sumat (i-1))
```

En imperativo:

```
int sumat (int x) {
    int s = 0, i = 0;
    while (i < x) {
        i = i + 1;
        s = s + i;
    }
    return s;
}
```

```
int sumat (int x) {
    int s = 0, i = 0;
    while (i < x) {
  // estado a
        i = i + 1;
        s = s + i;
  // estado b
    }
    return s;
}
```

Estados para $x == 4$

$i@a$	$s@a$	$i@b$	$s@b$
0	0	1	1
1	1	2	3
2	3	3	6
3	6	4	10

Observar que en cada paso: $0 \leq i \leq x$ y $s == \sum[0..i]$.
Estas dos condiciones se cumplen en estado a y estado b
(pero no en el medio)
Cuando $i == x$, la ejecución del ciclo termina.

Un ejemplo: demostremos que I es invariante

Sea $I : 0 \leq i \leq x \wedge s == \sum[0..i]$

```
int sumat (int x) {
    int s = 0; i = 0;
  //vale P_C : s == 0 ∧ i == 0
    while (i < x) {
    //invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
    //estado e1
        i = i + 1;
    //estado e2
        s = s + i;
    //estado e3
    }
  //vale Q_C : i == x ∧ s == ∑[0..x]
    return s;
}
```

Observemos que $P_C \rightarrow I$, por lo tanto I se cumple antes de entrar al ciclo.

El invariante es verdadero a lo largo de la ejecución del ciclo

La x no cambia porque es de entrada y no aparece en *modifica* ni en *local*.
Recordemos I : $0 \leq i \leq x \wedge s == \sum[0..i]$, $B : i < x$

```
  //estado e1
  //vale B ∧ I
  //implica 0 ≤ i < x ∧ s == ∑[0..i]
        i = i + 1;
  //estado e2
   //vale i == 1 + i@e1 ∧ s == s@e1
        s = s + i;
  //estado e3
  //vale i == i@e2 ∧ s == s@e2 + i@e2
  //implica i == i@e1 + 1 ∧ s == s@e2 + i@e1 + 1
```

porque $i@e3 == i@e2 == 1 + i@e1$.

```
  //implica 0 < i
```

porque $0 \leq i@e1$ y $i@e3 == i@e1 + 1$

```
  //implica i < 1 + x
```

porque $i@e1 < x,\ 1 + i@e1 < 1 + x,\ i@e3 == 1 + i@e1$

```
  //implica s == ∑[0..i]
```

porque $s@e3 == s@e2 + i@e2 == s@e1 + 1 + i@e1 ==$

$\sum[0..i@e1] + 1 + i@e1 == \sum[0..1 + i@e1] == \sum[0..i@e3]$ □

Terminación y correctitud de un ciclo respecto de una especificación

Requiere cinco expresiones del lenguaje de especificación:

una precondición P_C
una poscondición Q_C
un invariante I
una guarda B
una expresión variante v y una cota c.

Terminación y correctitud de un ciclo respecto de una especificación

```
  //vale P_C;
while (B) {
  //invariante I;
  //variante v; cota c
  cuerpo
}
  //vale Q_C;
```

Un ciclo termina si después de una cantidad finita de iteraciones se cumple la negación de la guarda.

Un ciclo es correcto respecto a la especificación si antes de su ejecución se cumple P_C y la ejecución del ciclo termina en un estado que cumple Q_C.

¿Cómo demostramos que el ciclo termina?

Conceptos similares que vimos para programación funcional.

Acotamos superiormente la cantidad de iteraciones restantes del ciclo mediante una expresión variante (v)

Es una expresión del lenguaje de especificación, de tipo Int

- definida a partir de las variables del programa
- debe decrecer estrictamente en cada iteración

```
//estado e;
//vale B ∧ I;
cuerpo
//vale v < v@e;
```

Damos una cota (c) (valor entero fijo, por ejemplo 0 o -8) tal que si es alcanzado por la expresión variante, está garantizado que la ejecución alcanza un estado donde vale la negación de la guarda, y por lo tanto sale del ciclo.

Ejemplo de expresión variante

```
int sumat (int x) {
    int s = 0; i = 0;
  //vale P_C : s == 0 ∧ i == 0
    while (i < x) {
    //invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
        i = i + 1;
        s = s + i;
    }
  //vale Q_C : i == x ∧ s == ∑[0..x]
    return s;
}
```

La expresión variante es un indicador de la distancia a la terminación del ciclo.

```
  //variante v : x − i, cota 0
```


Ejemplo de demostración de expresión variante decreciente.

Recordemos el programa `sumat` y la transformación de estados

```
//invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
//variante v : x − i
//estado e1
//vale B ∧ I
//implica 0 ≤ i < x ∧ s == ∑[0..i]
      i = i + 1;
//estado e2
//vale i == 1 + i@e1 ∧ s == s@e1
      s = s + i;
//estado e3
//vale i == i@e2 ∧ s == s@e2 + i@e2
```

Debemos ver que la expresión variante $v : x - i$ es decreciente.
Es decir, debemos ver que $v@e1 > v@e3$.
Sabemos que x no cambia, luego $x@e3 == x@e1 == pre(x)$. Luego,
$v@e1 == x@e1 - i@e1 == \text{pre}(x) - i@e1$ y
$v@e3 == x@e3 - i@3 == \text{pre}(x) - i@e3$.
Por la transformación de estados, $i@e3 = 1 + i@e1$. Por lo tanto,
$v@e1 == \text{pre}(x) - i@e1 > \text{pre}(x) - (1 + i@e1) == v@e3$.□

Teorema de terminación

Sea I el invariante de un ciclo con guarda B, v una expresión variante v (expresión entera decreciente) y c una cota tal que $(I \wedge v \leq c) \rightarrow \neg B$. Entonces el ciclo termina.

Demostración.

Sea v_j el valor que toma v en el estado que resulta de ejecutar el cuerpo del ciclo por j-ésima vez Dado que v es de tipo Int, para todo j, $v_j \in Int$.

Como v es estrictamente decreciente, $v_1 > v_2 > v_3 > \ldots$, existe un k tal que $v_k \leq c$. Dado que $(I \wedge v \leq c) \rightarrow \neg B$, en el estado alcanzado luego de k iteraciones vale $\neg B$. Por lo tanto el ciclo termina. □

¿Qué pasaría si v fuese de tipo Float? ¿Funcionaría esta demostración?

Expresión variante y cota

Sea v es una función variante y sea c su cota asociada.

- $v' = v - c$ es una función variante con cota asociada 0.

  Sin pérdida de generalidad, podemos suponer siempre una función variante y cota asociada 0.
- v es estrictamente decreciente en las sucesivas iteraciones del ciclo. En cada iteración decrece en 1 o más unidades. Por lo tanto el valor de la expresión variante no necesariamente es mide la cantidad de iteraciones (restantes) de la ejeccución del ciclo.

Observaciones sobre terminación

Sea el siguiente ciclo.

```
int dec1(int x) {
while (x > 0)
    x = x - 1;
return x;
}
```

Supongamos

$P_c : x \geq 0$.
$I : x \geq 0$
$v = x$ es estrictamente decreciente, cota $c = 0$.
Dado que $B : x > 0$, se cumple $v \leq c \rightarrow \neg B$
y por simple cálculo proposicional, $(I \wedge v \leq c) \rightarrow \neg B$.
Es decir, no fue necesario usar el invariante I.

Observaciones sobre terminación

```
int dec2(int x) {
while (x != 0)
    x = x - 1;
return x;
}
```

Supongamos
$P_c : x \geq 0$.
$I : x \geq 0$
$v = x$, cota $c = 0$.
Dado que $B : not(x == 0)$, $(I \wedge v \leq c) \rightarrow \neg B$

Notemos que en este caso sí necesitamos usar I
ya que $x \leq 0 \not\rightarrow x == 0$
pero $(x \geq 0 \wedge x \leq 0) \rightarrow x == 0$.

Teorema de Correctitud de un ciclo

Sea P_C, Q_C, I, B, v la especificación de un ciclo que termina.
Si se cumplen las relaciones de fuerza $P_C \to I$ y $(I \wedge \neg B) \to Q_C$ entonces el ciclo es correcto con respecto a su especificación.

Demostración.Debemos ver que para variables que satisfagan P_C, el estado alcanzado cuando el ciclo termina satisface Q_C.

```
  //estado e1
  //vale P_C;
while (B) {
  //vale I ∧ B;
  cuerpo
  //vale I;
}
  //estado e2
  //vale Q_C;
```

Supongamos que las variables en el estado e1 satisfacen P_C como $P_C \to I$, entonces en el estado e1 se cumple I.
Ejecutamos el ciclo (0 o más veces). Por definición de invariante (ver p. 16), en cada iteración, el invariante se restablece. Por hipótesis, el ciclo termina y en el estado e2 vale $\neg B$ Además, en el estado e2 vale I.
Como $(I \wedge \neg B) \to Q_C$ entonces en el estado e2 vale Q_C.□

Teorema del Invariante.Sea `while (B) { cuerpo }` un ciclo con guarda B, precondición P_C y poscondición Q_C. Sea I un predicado booleano, v una expresión variante, c una cota, y sean los estados e1 y e2 así

```
while (B) {
//estado e1
cuerpo
//estado e2}
```

Si valen

1. $P_C \to I$
2. $(I \wedge \neg B) \to Q_C$
3. el invariante se preserva en la ejecución del cuerpo, i.e. si $I \wedge B$ vale en el estado e1 entonces I vale en el estado $e2$
4. v es decreciente, i.e. $v@e1 > v@e2$
5. $(I \wedge v \leq c) \to \neg B$

entonces para cualquier valor de las variables del programa que haga verdadera P_C, el ciclo termina y hace verdadera Q_C, es decir, el ciclo es correcto para su especificación (P_C, Q_C).
Demostración.Inmediato del Teorema de Terminación (p. 21) y el Teorema de Correctitud (p. 25).□

Ejemplo de demostración de correctitud.
1. $P_C \to I$

```
int sumat (int x) {
    int s = 0, i = 0;
  //vale P_C : s == 0 ∧ i == 0
    while (i < x) {
  //invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
  //variante v : x − i
        i = i + 1;
        s = s + i;
    }
  //vale Q_C : i == x ∧ s == ∑[0..x]
    return s;
  //vale r == ∑[0..x]
}
```

Supongamos que vale P_C y veamos que vale I

1. $i == 0$ implica $0 \leq i$.
2. $i == 0$ y $x == 0$ implica $i \leq x$.
3. $s == 0$ y $i == 0$ implica $\sum[0..i] == \sum[0..0] == 0 == s$.

Por lo tanto, $i == 0 \wedge s == 0 \to \ 0 \leq i \leq x \wedge s == \sum[0..i]$.□

Ejemplo de demostración de correctitud.
2. $(I \wedge \neg B) \to Q_C$

```
int sumat (int x) {
    int s = 0, i = 0;
  //vale P_C : s == 0 ∧ i == 0
    while (i < x) {
  //invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
  //variante v : x − i
        i = i + 1;
        s = s + i;
    }
  //vale Q_C : i == x ∧ s == ∑[0..x]
    return s;
  //vale r == ∑[0..x]
}
```

Supongamos que vale $I \wedge \neg B$ y veamos que vale cada parte de Q_C.
Por I sabemos que $i \leq x$. Por $\neg B$ sabemos $i \geq x$. Entonces, $i == x$.
Por I sabemos $s == \sum[0..i]$, y recién mostramos que $i == x$, luego $s == \sum[0..x]$.□

Ejemplo de demostración. 3. El cuerpo preserva el invariante

La x no cambia porque es de entrada y no aparece en *modifica* ni en *local*.
Recordemos $I:\ 0 \leq i \leq x \wedge s == \sum[0..i]$, $B: i < x$

```
  //estado e1
  //vale B ∧ I
  //implica 0 ≤ i < x ∧ s == ∑[0..i]
        i = i + 1;
  //estado e2
   //vale i == 1 + i@e1 ∧ s == s@e1
        s = s + i;
  //estado e3
  //vale i == i@e2 ∧ s == s@e2 + i@e2
  //implica i == i@e1 + 1 ∧ s == s@e2 + i@e1 + 1
```

porque $i@e3 == i@e2 == 1 + i@e1$.

//implica $0 < i$

porque $0 \leq i@e1$ y $i@e3 == i@e1 + 1$

//implica $i < 1 + x$

porque $i@e1 < x,\ 1 + i@e1 < 1 + x,\ i@e3 == 1 + i@e1$

//implica $s == \sum[0..i]$

porque $s@e3 == s@e2 + i@e2 == s@e1 + 1 + i@e1 ==$

$\sum[0..i@e1] + 1 + i@e1 == \sum[0..1 + i@e1] == \sum[0..i@e3]$ □

Ejemplo de demostración. 4. La expresión variante es decreciente.

Recordemos la transformación de estados

```
  //estado e1
  //vale B ∧ I
  //implica 0 ≤ i < x ∧ s == ∑[0..i]
        i = i + 1;
  //estado e2
  //vale i == 1 + i@e1 ∧ s == s@e1
        s = s + i;
  //estado e3
  //vale i == i@e2 ∧ s == s@e2 + i@e2
```

Debemos ver que la expresión variante $v: x - i$ es decreciente.
Es decir, debemos ver que $v@e1 > v@e3$.
Sabemos que x no cambia, luego $x@e3 == x@e1 == pre(x)$. Luego,
$v@e1 == x@e1 - i@e1 == \text{pre}(x) - i@e1$ y
$v@e3 == x@e3 - i@3 == \text{pre}(x) - i@e3$.
Por la transformación de estados, $i@e3 = 1 + i@e1$. Por lo tanto,
$v@e1 == \text{pre}(x) - i@e1 > \text{pre}(x) - (1 + i@e1) == v@e3$.□

Ejemplo de demostración. 5. $(I \wedge v \leq c) \rightarrow \neg B$

```
int sumat (int x) {
    int s = 0, i = 0;
  //vale P_C : s == 0 ∧ i == 0
    while (i < x) {
  //invariante I : 0 ≤ i ≤ x ∧ s == ∑[0..i]
  //variante v : x - i
        i = i + 1;
        s = s + i;
    }
  //vale Q_C : i == x ∧ s == ∑[0..x]
    return s;
  //vale r == ∑[0..x]
}
```

Supongamos que vale $I \wedge v \leq c$. Debemos ver que vale $\neg B$.
Esta vez no hace falta usar I.
Como la cota es 0, $v \leq c$ es equivalente a $x - i \leq 0$,
que a su vez es equivalente a $x \leq i$;
y ésto es exactamente $\neg B$, ya que $B: i < x$.□