Teórica Imperativo 2
Estructuras de control

ALGORITMOS Y ESTRUCTURAS DE DATOS I

Esquema

- Instrucción de asignación
  - P { v = e } Q

```
// estado 1; vale P: true
v:=e;
// vale Q: v == e@1 && para toda vi: vi == vi@1
```

- Instrucciones de Control de Flujo
  - P { if (B) *uno* else *dos* } Q
  - P { while (B) *cuerpo* } Q
- Demostraciones
  - Correctitud
    - **Invariantes de Ciclos**
  - Terminación
    - Expresión Variante

Condicionales

- if (B) *uno* else *dos*
  - B tiene que ser una expresión booleana (verdadera o falsa) sin efectos secundarios (no tiene que modificar el estado)
  - *uno* y *dos* son instrucciones
    - En particular, pueden ser bloques (entre llaves)
- La semántica requiere
  - Una precondición del condicional P
  - Una poscondición del condicional Q que se cumpla en ambos casos
- Si sé que

```
//vale P && B;        //vale P && ¬B;
uno                   dos
//vale Q;             //vale Q;
```

- Puedo deducir

```
//vale P;
if (B) uno else dos
//vale Q;
```


Ejemplo de condicional

```
//problema max(x, y: Int) = result: Int {
//   asegura Q: (x > y ∧ result == x) ∨ (x ≤ y ∧ result == y);
//}
int max(int x, int y) {
   int m = 0;
   // vale Pif: m == 0;
   if (x > y)
        m = x;
   else
        m = y;
   // vale Qif: (x > y ∧ m == x) ∨ (x ≤ y ∧ m == y);
   return m;
   // vale Q;
}
```

- Demostración
  - Tenemos que ver que por las dos ramas llegamos a Qif
    - /*vale m == 0 ∧ x > y;*/ m = x; /*vale Qif;*/
    - /*vale m == 0 ∧ x ≤ y;*/ m = y; /*vale Qif;*/

Demostración del condicional

- Demuestro cada rama **fuera** del condicional
- `Qif: (x > y ∧ m == x) ∨ (x ≤ y ∧ m == y);`
- Rama `true`

```
    //vale m == 0 ∧ x > y;
m = x;
    //vale x > y ∧ m == x;
    //implica (x > y ∧ m == x) ∨ (x ≤ y ∧ m == y);
```

- Rama `false`

```
    //vale m == 0 ∧ x ≤ y;
m = y;
    //vale x ≤ y ∧ m == y;
    //implica (x > y ∧ m == x) ∨ (x ≤ y ∧ m == y);
```

- Pudimos llegar a `Qif` por las dos ramas,
  - entonces `Qif` es la poscondición del condicional

Ciclos

- `while (B)` *cuerpo*
- `B:` expresión booleana, sin efectos colaterales
  - También se la llama *guarda*
- *cuerpo* es una instrucción
  - En particular, puede ser un bloque (entre llaves)
  - Se repite mientras `B` valga
    - Cero o más veces
    - Si es una cantidad finita, el programa **termina**
      - Si es > 0, alguna de las ejecuciones tiene que hacer `B` falsa
      - Y el estado final de esa ejecución será el **estado final del ciclo**

Ejemplo

```
//problema sumax(x: Int) = result: Int {
//   requiere P: x >= 0;
//   asegura result == sum([0..x])
//}
```

Sabemos implementarlo en funcional…

```
sumax:: Int -> Int
sumax 0 = 0;
suma (x+1) = (x+1) + (sumax x)
```

x	s
0	0
1	1
2	3
3	6

Ejemplo

```
//problema sumax(x: Int) = result: Int {
//   requiere P: x >= 0;
//   asegura result == sum([0..x])
//}
```

```
int sumax (int x) {
   int s = 0, i = 0;
   while (i < x) {



   }
   return s;
}
```

i	s
0	0
1	1
2	3
3	6

Ejemplo

```
//problema sumax(x: Int) = result: Int {
//   requiere P: x >= 0;
//   asegura result == sum([0..x])
//}
```

```
int sumax (int x) {
   int s = 0, i = 0;
   while (i < x) {

      i = i + 1;
      s = s + i;

   }
   return s;
}
```

i	s
0	0
1	1
2	3
3	6

Ejemplo

```
//problema sumax(x: Int) = result: Int {
//   requiere P: x >= 0;
//   asegura result == sum([0..x])
//}
```

```
int sumax (int x) {
   int s = 0, i = 0;
   while (i < x) {
      // estado 1
      i = i + 1;
      s = s + i;
      // estado 2
   }
   return s;
}
```

i@e1	s@e1	i@e2	s@e2
0	0	1	1
1	1	2	3
2	3	3	6
3	6	4	10

s == sum([0..i]) ∧ 0 <= i <= x

Semántica de ciclos

- La semántica requiere tres expresiones del lenguaje de especificación
  - Una **precondición** P
  - Una **poscondición** Q
  - Un **invariante** I
  - Un **guarda** B
  - Una **expresión variante** v y una **cota** c

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


Semántica de ciclos

- Invariante
  - Condición cuya veracidad es preservada por el cuerpo del ciclo
  - Vale en cada iteración (hipotesis inductiva!)
  - No hay forma algorítmica de encontrarlo
  - Conviene darlo al escribir el ciclo, porque encierra la idea del programador o diseñador

Ejemplo

```
//problema sumax(x: Int) = res: Int {
//   requiere x >= 0;
//   asegura res == sum([0..x]);
//}

int sumax (int x) {
   int s = 0, i = 0;
      //vale P: x >= 0 && s == i == 0;
   while (i < x) {
      //invariante I: 0 <= i <= x && s == sum([0..i]);
       i = i + 1;
       s = s + i;
   }
      //vale Q: s == sum([0..x]);
   return s;
      //vale res == sum([0..x]);
}
```


Ejemplo de invariante

- Factorial

```
//problema Factorial(n: Int) = result: Int {
//   requiere R: n >= 0;
//   asegura result == prod([1..n])
//}
int Factorial(int n) {
   int f = 1;
      //vale R ∧ f == 1;
   int i = 1;
      //vale Pc: R ∧ f == i == 1;
   while (i < n) {
      //invariante 1 <= i <= n ∧ f == prod([1..i]);
      i = i + 1;
      f = f * i;
   }
      //vale Qc: i==n && f == prod([1..n]);
   return f;
      //vale result == prod([1..n]);
}
```


Teorema

- Si sé que

$$\underbrace{P \rightarrow I}_{\text{El invariante vale al comienzo}} \qquad \underbrace{(\neg B \wedge I) \rightarrow Q}_{\text{La poscondición vale al final}}$$

```
//vale B ∧ I;
cuerpo
//vale I;
```

$\Big\}$ El cuerpo preserva el invariante

- **Si el ciclo termina**, puedo deducir

```
// vale P;
while (B) cuerpo
//vale Q;
```


Demostración

1) P → I
2) //vale B ∧ I;
   cuerpo
   //vale I;
3) (¬B ∧ I) → Q

- Nombramos estados

```
//estado e0;
while (B) {
   cuerpo
   //estado ej; (para j == 1...)
}
//estado final;
```

- Queremos ver que vale `Q@final==true`
- Por hipótesis, `P@`e_0`==true` y `P → I`,
  - entonces `I@`e_0`==true`
- Como *cuerpo* preserva `I`: `I@`e_j`==true` para todo j (inducción)
- Como el ciclo termina, hay un k tal que `¬B@`e_k`==true`
  - `Y final =` e_k
- Y, como e_k es uno de los e_j, entonces `I@`e_k`==I@final==true`
- Dado que `(¬B ∧ I) → Q`,
  - concluimos `Q@final==true`

Utilizando el teorema del invariante

- Dado un ciclo
  - Precondición: P y Poscondición Q
  - Invariante: I
  - Guarda: B y su cuerpo
- Para ver que es correcto (si termina) con respecto a P y Q basta con probar las **hipótesis** del teorema!
  1) $P \rightarrow I$
  2) //vale $B \wedge I$;
     cuerpo
     //vale I;
  3) $(\neg B \wedge I) \rightarrow Q$

Ejemplo

```
//problema sumax(x: Int) = res: Int {
//   requiere x >= 0;
//   asegura res == sum([0..x]);
//}

int sumax (int x) {
   int s = 0, i = 0;
      //vale P: x >= 0 && s == i == 0;
   while (i < x) {
      //invariante I: 0 <= i <= x && s == sum([0..i]);
      i = i + 1;
      s = s + i;
   }
      //vale Q: s == sum([0..x]);
   return s;
      //vale res == sum([0..x]);
}
```


Terminación

- La semántica que vimos depende de la suposición de que el ciclo termina
- Es una suposición muy fuerte
- Los **ciclos** son las instrucciones que pueden hacer que un programa **se cuelgue**
- Si queremos escribir programas correctos, tenemos que **garantizar que todos sus ciclos terminen**

Ejemplo de terminación

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

- Difieren en la guarda
  - Si x no es negativo
    - Ambos devuelven 0
    - Iteran la misma cantidad de veces (x)
  - De lo contrario
    - `dec(x)` devuelve x
    - pero `dec2(x)` nunca sale del ciclo
- Necesitamos herramientas para diferenciarlos

Expresión variante y cota

- Similares a conceptos que vimos para programación funcional
- Para hablar de la cantidad de veces que se ejecuta un ciclo
  - **Expresión variante** (V)
    - Es una expresión del lenguaje de especificación, de tipo `Int`
    - Usa variables del programa
  - **Cota** (c)
    - Valor que, si es alcanzado o pasado por la expresión variante, garantiza que la ejecución sale del ciclo
    - `I && v ≤ c → ¬B`

Expresión variante

- Sean
  - Un ciclo `while (B)` *cuerpo*
  - Una expresión variante `v: Int`
- Decimos que
  - v es *monótona decreciente en la ejecución del ciclo*
- Si y solo si

```
    //estado a;
cuerpo
    //vale v < v@a;
```

- Sin pérdida de generalidad, podemos considerar la cota 0
  - Si existe una expresión variante v monótona decreciente para un ciclo y una cota c que garantiza la salida del ciclo, entonces existe otra expresión v', también monótona decreciente, con cota 0
  - Basta con definir `aux v' = v - c;`

Ejemplo de expresión variante

```
int dec(int x) {
   while (x > 0)
      //variante x;
      x = x - 1;
   return x;
}
```

- Cota 0
  - Si fuera otro valor, se aclara: `variante(3) x+3;`
- Veamos que es decreciente en el ciclo

```
    //estado a;
x = x - 1;
    //vale x == x@a - 1 < x@a;
```

- Veamos que si alcanza (o pasa) la cota, el ciclo termina
  - Supongamos x ≤ 0
  - Entonces no se cumple la guarda y el ciclo termina

No terminación

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

- `variante x;` es decreciente, pero la cota 0 no garantiza terminación
  - Para `x < 0` no se falsea la guarda
- Probemos otras
  - `variante x * β(x ≥ 0);`
    - Es decir
      - `variante x` si x ≥ 0
      - `variante 0` si no
    - Pero para x < 0 la expresión es constante, no decrece

Demostración de no terminación

- ¿Entonces el ciclo no termina?
  - Lo único que pasó es que no logramos encontrar la expresión
  - Podría haber una que sirviera
  - Aunque en este caso no hay ninguna (el ciclo no termina)
- Puede ser difícil demostrar la no existencia de expresiones estrictamente decrecientes y acotadas
  - No existe un método algorítmico para demostrar si las hay

Terminación y correctitud de un ciclo

- Sean
  - Un ciclo `while (B)` *cuerpo*
  - Una expresión variante `v: Int`
  - Una cota `c`
  - Expresiones booleanas `P`, `Q` e `I`
- Si se cumplen las siguientes condiciones,
  1. `P → I`
  2. `/*vale B ∧ I;*/` *cuerpo* `/*vale I;*/`
  3. `/*estado a; vale B ∧ I;*/` *cuerpo* `/*vale v < v@a;*/`
  4. `(I ∧ v ≤ c) → ¬B`
  5. `(¬B ∧ I) → Q`

  El ciclo **termina** y es **correcto** respecto de `P` y `Q`

Teorema del invariante

Sean:
- Un ciclo `while (B)` *cuerpo*
- Una expresión variante `v: Int`
- Una cota `c`
- Expresiones booleanas `P`, `Q` e `I`

Si se cumplen las siguientes condiciones
1. `P → I`
2. /* vale `B ∧ I`;*/ *cuerpo* /*vale `I`;*/
3. /*estado a; vale `B ∧ I`;
   */ *cuerpo* /* vale v < v@a;*/
4. `(I ∧ v ≤ c) → ¬B`
5. `(¬B ∧ I) → Q`

Entonces, el ciclo termina y es correcto respecto de P y Q

- Demostración
  - 1 y 2 garantizan que `I` sea un invariante del ciclo
  - 3 asegura que **V** sea monótona decreciente en la ejecución del ciclo
  - 4 garantiza que el ciclo termine
  - 5 asegura que la poscondición sea verdadera en el estado final

Ejemplo completo

```
//problema sumax(x: Int) = res: Int {
//   requiere x >= 0;
//   asegura res == sum([0..x]);
//}
```

Ejemplo completo

```
//problema sumax(x: Int) = res: Int {
//   requiere x >= 0;
//   asegura res == sum([0..x]);
//}

int sumax (int x) {
   int s = 0, i = 0;
      //vale P:
   while (i < x) {
      //invariante I:
      //variante v:
       i = i + 1;
       s = s + i;
   }
      //vale Q:
   return s;
      //vale res == sum([0..x]);
}
```


Ejemplo completo

```
//problema sumax(x: Int) = res: Int {
//   requiere x >= 0;
//   asegura res == sum([0..x]);
//}

int sumax (int x) {
   int s = 0, i = 0;
      //vale P: x >= 0 && s == i == 0;
   while (i < x) {
      //invariante I: 0 <= i <= x && s == sum([0..i]);
      //variante v: x - i;
       i = i + 1;
       s = s + i;
   }
      //vale Q: s == sum([0..x]);
   return s;
      //vale res == sum([0..x]);
}
```


1 y 2. I es invariante

- El invariante vale al principio
  - Queremos ver que P → I
    - `P: x >= 0 ∧ s == i == 0`
    - `I: 0 <= i <= x ∧ s == sum([0..i])`
  - Si P `==` `true`, entonces (`o <= x ∧ 0 == sum([0..0]) == true`)
- El cuerpo preserva el invariante
  - qvq `/*vale B ∧ I;*/ i = i + 1; s = s + i; /*vale I;*/`

```
    //estado 1;
    //vale B ∧ I;
    //implica 0 ≤ i < x ∧ s == sum([0..i]);
 i = i + 1
    //estado 2;
    //vale i == i@1 + 1 ∧ s == s@1;
 s = s + i;
    //estado 3
    //vale i == i@2 ∧ s == s@2 + i;
    //implica i == i@1 + 1 ∧ s == s@2 + i;
    //implica s == s@1 + (i@1 + 1) == sum([0..i@1]) + (i@1 + 1);
    //implica s == sum([0..(i@1 + 1)]) == sum([0..i]);
    //implica 0 < i <= x ∧ s = sum([0..i]); (porque i@1<x ➔i@i1+1<=x)
    //implica I  (porque 0<i ➔ 0<=i)
```

  - qed

3 y 4. v es decreciente y 0 es cota

- `v=x-i` es estrictamente decreciente en la ejecución del ciclo:
  `/*estado 1; vale B ∧ I;*/ i = i+1; s = s+i; /*estado 3; vale v < v@1;*/`
    - Vimos que `/*estado 1;*/ i = i+1; s = s+i; /*estado 3; vale i == i@1 + 1 && …;*/`
    - `vale v@1 == x - i@1;`
    - `vale v@3 == x - i@3 == x - (i@1 + 1) == x - i@1 - 1 < v@1;`
  - qed
- Si v pasa la cota, el ciclo termina
  - `(I ∧ v ≤ 0) → ¬B`
    - ¬B es `i ≥ x`
    - `v == x - i`
    - `v ≤ 0 → x-i ≤ 0 ↔ x ≤ i ↔ i ≥ x`
  - qed

5.Poscondición

- La poscondición vale al final
  - ¬B ∧ I → Q
    - `Q: s == sum([0..x])`
    - `¬B ∧ I ⇔ i ≥ x ∧ 0 ≤ i ≤ x ∧ s == sum([0..i])`
    - `(i ≥ x ∧ 0 ≤ i ≤ x) → i == x`
    - `implica s == sum([0..x])`
  - qed

Lo que viene…

- Operar con Arreglos
  - Pasaje por referencia
  - Especificar "efectos"
- Algoritmos sencillos
  - Búsqueda Lineal
  - Búsqueda Binaria
- Demostraremos su correctitud