Programación 2

Lección 12. Análisis de la corrección de un algoritmo: segunda parte

1. Corrección de instrucciones en las que se invoca a una función
2. Corrección de una instrucción de invocación a una función
   - Aplicación a un ejemplo
3. Corrección de una instrucción con una expresión con invocaciones a funciones
   - Aplicación a un ejemplo

1. Corrección de instrucciones en las que se invoca a una función

Consideraremos la función **`acción(p)`** que no devuelve ningún valor al ser invocada:

```
// Pre: P(p)
// Post: Q(p)
void acción (tipoParámetro p);
```

Y la función **`calcular(p)`** que devuelve un valor de tipo **`tipoResultado`** al ser invocada:

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

Consideraremos la función **`acción(p)`** que no devuelve ningún valor al ser invocada:

```
// Pre: P(p)
// Post: Q(p)
void acción (tipoParámetro p);
```

Y analizaremos en el siguiente apartado la corrección de una instrucción de invocación a la función anterior:

```
// R
acción(E);
// S
```

Consideraremos la función **`calcular(p)`** que devuelve un valor de tipo al **`tipoResultado`** ser invocada:

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

Y analizaremos en el apartado §3 la corrección de las siguientes instrucciones con invocaciones a la función anterior:

```
// R
x = x + calcular(E);
// S
```

```
// R
return calcular(E);
// S
```

2. Corrección de una instrucción de invocación a una función

Consideremos la función **`acción(p)`** que no devuelve ningún valor al ser invocada:

```
// Pre: P(p)
// Post: Q(p)
void acción (tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción de invocación sea correcta?

```
// R
acción(E);
// S
```

```
// Pre: P(p)
// Post: Q(p)
void acción (tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción de invocación sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E

```
// R  ⇒  Dom(E)
acción(E);
// S
```

```
// Pre: P(p)
// Post: Q(p)
void acción (tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción de invocación sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E
2. Cualquier predicado I que se satisfaga antes de la ejecución de la instrucción, y en el que no intervengan datos modificados por la instrucción, se satisfará también después.

```
// R  ⇒  Dom(E)
// R  ⇒  I
acción(E);
// I
// S
```

```
// Pre: P(p)
// Post: Q(p)
void acción (tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción de invocación sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E
2. Cualquier predicado I que se satisfaga antes de la ejecución de la instrucción, y en el que no intervengan datos modificados por la instrucción, se satisfará también después
3. La satisfacción de la precondición de la invocación, P(E), garantiza la satisfacción de la postcondición, Q(E)

```
// R  ⇒  Dom(E) ∧ I
// R  ⇒  P(E)
acción(E);
// I ∧ Q(E)
// S
```

```
// Pre: P(p)
// Post: Q(p)
void acción (tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción de invocación sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E
2. Cualquier predicado I que se satisfaga antes de la ejecución de la instrucción, y en el que no intervengan datos modificados por la instrucción, se satisfará también después
3. La satisfacción de la precondición de la invocación, P(E), garantiza la satisfacción de la postcondición, Q(E)
4. El predicado $I \wedge Q(E)$ debe garantizar la satisfacción de la postcondición S

```
// R  ⇒  Dom(E) ∧ I ∧ P(E)
acción(E);
// I ∧ Q(E)  ⇒  S
// S
```

1º

2º

Aplicación a un ejemplo. Consideremos la función **`permutar()`** que no devuelve ningún valor al ser invocada:

```
// Pre: xx = A ∧ yy = B
// Post: xx = B ∧ yy = A
void permutar (int& xx, int& yy);
```

Debemos demostrar la corrección de la instrucción:

```
// x ≥ 100 ∧ y ≤ 0
permutar(x,y);
// x < y
```

Aplicación a un ejemplo (continuación)

```
// Pre: xx = A ∧ yy = B
// Post: xx = B ∧ yy = A
void permutar (int& xx, int& yy);
```

Deducimos un predicado I que sea invariante respecto de la instrucción y que nos facilite la demostración de su corrección:

```
// x ≥ 100 ∧ y ≤ 0 ≡ x ≥ 100 ∧ y ≤ 0 ∧ x = A ∧ y = B
//  ⇒  I
permutar(x,y);
// I
// x < y
```

Aplicación a un ejemplo (continuación)

```
// Pre: xx = A ∧ yy = B
// Post: xx = B ∧ yy = A
void permutar (int& xx, int& yy);
```

Escribimos el predicado **I ≡ A ≥ 1000 ∧ B ≤ 0**, que es invariante respecto de la instrucción permutar(x,y):

```
// x ≥ 100 ∧ y ≤ 0
// x ≥ 100 ∧ y ≤ 0 ≡ x ≥ 100 ∧ y ≤ 0 ∧ x = A ∧ y = B
// ⇒
// A ≥ 100 ∧ B ≤ 0
permutar(x,y);
// A ≥ 100 ∧ B ≤ 0
// x < y
```

Aplicación a un ejemplo (continuación)

```
// Pre: xx = A ∧ yy = B
// Post: xx = B ∧ yy = A
void permutar (int& xx, int& yy);
```

Se satisface la precondición **x = A ∧ y = B** de la invocación permutar(x,y), por lo tanto también se satisfará su postcondición:

```
// x ≥ 100 ∧ y ≤ 0
// x ≥ 100 ∧ y ≤ 0 ≡ x ≥ 100 ∧ y ≤ 0 ∧ x = A ∧ y = B
// ⇒
// x = A ∧  y = B
permutar(x,y);
// A ≥ 100 ∧ B ≤ 0 ∧ x = B ∧ y = A
// x < y
```

Aplicación a un ejemplo (continuación)

```
// Pre: xx = A ∧ yy = B
// Post: xx = B ∧ yy = A
void permutar (int& xx, int& yy);
```

El predicado **A ≥ 100 ∧ B ≤ 0 ∧ x = B ∧ y = A** es más fuerte que el predicado **x < y**

```
// x ≥ 100 ∧ y ≤ 0 ≡ x ≥ 100 ∧ y ≤ 0 ∧ x = A ∧ y = B
// ⇒                                                    1º
// x = A ∧ y = B
permutar(x,y);                                          ✓
// A ≥ 100 ∧ B ≤ 0 ∧ x = B ∧ y = A
// ⇒                                                    2º
// x < y

// x < y
```

3. Corrección de una instrucción con una expresión con invocaciones a funciones

Consideramos la función **`calcular(p)`** que devuelve un valor de tipo **`tipoResultado`** al ser invocada:

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción, con una invocación a una función, sea correcta?

```
// R
x = x + calcular(E);
// S
```

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción, con una invocación a una función, sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E

```
// R ⇒ Dom(E)
x = x + calcular(E);
// S
```

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción, con una invocación a una función, sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E
2. El predicado R debe garantizar la satisfacción de la precondición `P(E)` de la invocación `calcular(E)`

```
// R  ⇒  Dom(E) ∧ P(E)
x = x + calcular(E);
// S
```

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción, con una invocación a una función, sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E
2. El predicado R debe garantizar la satisfacción de la precondición `P(E)` de la invocación `calcular(E)`
3. Se satisface la postcondición `Q(E,calcular(E))` de la invocación `calcular(E)`

```
// R ⇒ Dom(E) ∧ P(E)
// R ∧ Q(E,calcular(E))
x = x + calcular(E);
// S
```

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción, con una invocación a una función, sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E
2. El predicado R debe garantizar la satisfacción de la precondición `P(E)` de la invocación `calcular(E)`
3. Se satisface la postcondición `Q(E,calcular(E))` de la invocación `calcular(E)`
4. El predicado R $\wedge$ `Q(E,calcular(E))` garantiza la satisfacción de la precondición mas débil $\boldsymbol{S_x^{x+calcular(E)}}$ de la asignación `x = x + calcular(E)`

```
// R ⇒ Dom(E) ∧ P(E)
// R ∧ Q(E,calcular(E)) ⇒ S_x^{x+calcular(E)}
x = x + calcular(E);
// S
```

1º
2º

Consideramos la función **`calcular(p)`** que devuelve un valor de tipo **`tipoResultado`** al ser invocada:

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción de devolución de valor sea correcta?

```
// R
return calcular(E);
// S
```

Consideramos la función **`calcular(p)`** que devuelve un valor de tipo **`tipoResultado`** al ser invocada:

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción, con una invocación a una función, sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E

```
// R ⇒ Dom(E)
return calcular(E);
// S
```

Consideramos la función **`calcular(p)`** que devuelve un valor de tipo **`tipoResultado`** al ser invocada:

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción, con una invocación a una función, sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E
2. El predicado R debe garantizar la satisfacción de la precondición `P(E)` de la invocación `calcular(E)`

```
// R ⇒ Dom(E) ∧ P(E)
return calcular(E);
// S
```

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción, con una invocación a una función, sea correcta?

1. Que se garantice la evaluación sin errores de la expresión `E`
2. El predicado R debe garantizar la satisfacción de la precondición `P(E)` de la invocación `calcular(E)`
3. Se satisface la postcondición `Q(E,calcular(E))` de la invocación `calcular(E)`

```
// R ⇒ Dom(E) ∧ P(E)
// R ∧ Q(E,calcular(E))
return calcular(E);
// S
```

```
// Pre: P(p)
// Post: Q(p,calcular(p))
tipoResultado calcular (const tipoParámetro p);
```

¿Qué condiciones se exigen para que la siguiente instrucción, con una invocación a una función, sea correcta?

1. Que se garantice la evaluación sin errores de la expresión E
2. El predicado R debe garantizar la satisfacción de la precondición `P(E)` de la invocación `calcular(E)`
3. Se satisface la postcondición `Q(E,calcular(E))` de la invocación `calcular(E)`
4. El predicado `R` $\wedge$ `Q(E,calcular(E))` garantiza la satisfacción de la precondición mas débil $S^{calcular(E)}_{nombreFunción(\dots)}$ de la instrucción `return calcular(E);`

```
// R ⇒ Dom(E) ∧ P(E)                                          1º
// R ∧ Q(E,calcular(E)) ⇒ S^{calcular(E)}_{nombreFunción(...)}  2º
return calcular(E);
// S
```

Aplicación a un ejemplo.

```
// Pre: n ≥ 0
// Post: factorial(n) = (Πα∈[1,n]. α)
long int factorial (const int n);
```

Demostrar formalmente la corrección del siguiente código:

```
// n > 0 ∧ m ≥ 0 ∧ n ≥ m
long int c = factorial(n) / (factorial(m) * factorial(n-m));
// c = (n m)
```

Aplicación a un ejemplo.

```
// Pre: n ≥ 0
// Post: factorial(n) = (Πα∈[1,n]. α)
long int factorial (const int n);
```

Demostrar formalmente la corrección del siguiente código:

1. Se satisfacen las condiciones de dominio de las expresiones de las tres invocaciones

```
// n > 0 ∧ m ≥ 0 ∧ n ≥ m  ⇒  Dom(n) ≡ cierto
// n > 0 ∧ m ≥ 0 ∧ n ≥ m  ⇒  Dom(m) ≡ cierto
// n > 0 ∧ m ≥ 0 ∧ n ≥ m  ⇒  Dom(n-m) ≡ cierto
long int c = factorial(n) / (factorial(m) * factorial(n-m));
// c = (n m)
```

1º

Aplicación a un ejemplo.

```
// Pre: n ≥ 0
// Post: factorial(n) = (Πα∈[1,n]. α)
long int factorial (const int n);
```

Demostrar formalmente la corrección del siguiente código:

1. Se satisfacen las condiciones de dominio de las expresiones de las tres invocaciones
2. Se satisfacen sus tres precondiciones

```
// n > 0 ∧ m ≥ 0 ∧ n ≥ m  ⇒  n ≥ 0
// n > 0 ∧ m ≥ 0 ∧ n ≥ m  ⇒  m ≥ 0
// n > 0 ∧ m ≥ 0 ∧ n ≥ m  ⇒  n - m ≥ 0
long int c = factorial(n) / (factorial(m) * factorial(n-m));
// c = (n m)
```

2º

```
// Pre: n ≥ 0
// Post: factorial(n) = (Πα∈[1,n]. α)
long int factorial (const int n);
```

Demostrar formalmente la corrección del siguiente código:

1. Se satisfacen las condiciones de dominio de las expresiones de las tres invocaciones
2. Se satisfacen sus tres precondiciones y, por lo tanto, sus tres postcondiciones

```
// n > 0 ∧ m ≥ 0 ∧ n ≥ m

// n > 0 ∧ m ≥ 0 ∧ n ≥ m ∧
// factorial(n) = (Πα∈[1,n]. α) ∧
// factorial(m) = (Πα∈[1,m]. α) ∧
// factorial(n-m) = (Πα∈[1,n-m]. α)
long int c = factorial(n) / (factorial(m) * factorial(n-m));
// c = (n m)
```

3º

Demostrar formalmente la corrección del siguiente código:

1. Se satisfacen las condiciones de dominio de las expresiones de las tres invocaciones
2. Se satisfacen sus tres precondiciones y, por lo tanto, sus tres postcondiciones
3. ¿Se satisface la precondición más débil de la instrucción de asignación?

```
// n > 0 ∧ m ≥ 0 ∧ n ≥ m
// n > 0 ∧ m ≥ 0 ∧ n ≥ m ∧ factorial(n) = (Πα∈[1,n]. α) ∧
//                         factorial(m) = (Πα∈[1,m]. α) ∧
//                         factorial(n-m) = (Πα∈[1,n-m]. α)
// ¿ ⇒ ?
// factorial(n) / (factorial(m) * factorial(n-m)) = (n m)      4º
long int c = factorial(n) / (factorial(m)*factorial(n-m));
// c = (n m)
```

Demostrar formalmente la corrección del siguiente código:

1. Se satisfacen las condiciones de dominio de las expresiones de las tres invocaciones
2. Se satisfacen sus tres precondiciones y, por lo tanto, sus tres postcondiciones
3. Se satisface la precondición más débil de la instrucción de asignación

```
// n > 0 ∧ m ≥ 0 ∧ n ≥ m
// n > 0 ∧ m ≥ 0 ∧ n ≥ m ∧ factorial(n) = (Πα∈[1,n]. α) ∧
//                         factorial(m) = (Πα∈[1,m]. α) ∧
//                         factorial(n-m) = (Πα∈[1,n-m]. α)
// ⇒
// factorial(n) / (factorial(m) * factorial(n-m)) = (n m)        4º
long int c = factorial(n) / (factorial(m)*factorial(n-m));
// c = (n m)
```