Inmersiones

Josefina Sierra Santibáñez

22 de abril de 2018

Inmersión o Generalización de una Función

Una **inmersión** de una función es una generalización de dicha función donde se introducen parámetros adicionales, resultados adicionales o ambas cosas.

Existen varios tipos de inmersiones.

- **Inmersión de especificación**: Se añaden nuevos parámetros para realizar un diseño recursivo de una función cuya especificación inicial no permite implementarla recursivamente de manera natural.

En este capítulo veremos dos tipos de inmersiones de especificación:

- Inmersión por **relajación de la postcondición**
- Inmersión por **refuerzo de la precondición**

Todos las ejemplos de este bloque de transparencias son diseños recursivos de funciones realizados mediante la técnica de **inmersión por especificación**.

Inmersiones de Eficiencia

Una **inmersión de eficiencia** es una generalización de una función en la que los parámetros o resultados adicionales permiten pasar información de una llamada recursiva a otras, y de ese modo evitar la repetición innecesaria de ciertos cálculos.

Existen varios tipos de inmersiones de eficiencia.

- **Inmersión por Datos**: Se añaden nuevos datos (parámetros de entrada) con la intención de pasar información desde una llamada recursiva a la siguiente.
- **Inmersión por Resultados**: Se obtiene información de la siguiente llamada recursiva (a partir de resultados o parámetros de salida) para utilizarla en la llamada actual.
- **Inmersión por Datos y Resultados**: Combina las dos formas de inmersión anteriores.

Una **inmersión de eficiencia** es una inmersión en la que los datos (o resultados) que pasa una llamada recursiva a la siguiente (o a la anterior) permiten ahorrar calculos en la llamada que los recibe.

Ejemplos de Inmersión por Relajación de la Postcondición

El objetivo es obtener un diseño recursivo de

```
int suma(const vector<int>& v) {
/* Pre: cierto */
/* Post: el resultado es la suma de los elementos de v */
  return i_suma(v, v.size());
}
```

Para lograrlo introducimos una función auxiliar recursiva que contiene un parámetro adicional y cuya postcondición es más débil que la de la función original.

```
int i_suma(const vector<int>& v, int i);
/* Pre: 0 ≤ i ≤ v.size() */
/* Post: el resultado es la suma de los
   elementos de v[0...i-1] */
```

La implementación de estas funciones está en el archivo `suma_vec_r.cc`

Ejemplos de Inmersión por Refuerzo de la Precondición

El objetivo es obtener un diseño recursivo de

```
int suma(const vector<int>& v) {
/* Pre: cierto */
/* Post: el resultado es la suma de los elementos de v */
  return i_suma_2(v,0,0);
}
```

Introducimos una función auxiliar recursiva que contiene dos parámetros adicionales cuyo valor es una posición `i` y la suma parcial de los elementos del subvector `v[0...i-1]`. La introducción de dichos parámetros supone añadir a la precondición una versión débil de la postcondición, de manera que a partir de la precondición reforzada sea más fácil alcanzar la postcondición (ver `suma_vec_r_2.cc`).

```
int i_suma_2(const vector<int>& v, int i,
             int s_i);
/* Pre: 0 ≤ i ≤ v.size(),
s_i es la suma de los elementos de v[0...i-1] */
/* Post: el resultado es la suma de los elementos de v */
```

Búsqueda lineal (inmersión por Relajación de Post)

El objetivo es obtener un diseño recursivo de

```
bool cerca(const list<int>& s, int x) {
/* Pre: cierto */
/* Post: El resultado indica si x está en s. */
  return cerca_r(s, x, s.end());
}
```

Para lograrlo introducimos una función auxiliar recursiva que contiene un parámetro adicional y cuya postcondición es más débil que la de la función original.

```
bool cerca_r(const list<int> &s, int x,
     list<int>::const_iterator it);
/* Pre: it apunta a s.end() o a un elemento de s. */
/* Post: El resultado indica si x es un elemento
de la sublista comprendida entre s.begin() y el
elemento anterior al que apunta it. */
```

La implementación de estas acciones está en `buscaRec.cc`.

Búsqueda lineal (inmersión por refuerzo de la precondición)

El objetivo es obtener un diseño recursivo de

```
bool cerca(const list<int>& s, int x) {
/* Pre: cierto */
/* Post: El resultado indica si x está en s. */
  return cerca_r_2(s, x, s.begin());}
```

Introducimos una función auxiliar recursiva que contiene un parámetro adicional que delimita la parte de s donde ya hemos realizado la búsqueda. La introducción de dicho parámetro permite añadir a la precondición una versión débil de la postcondición, de manera que a partir de la precondición reforzada sea más fácil alcanzar la postcondición (ver `buscaRec2.cc`).

```
bool cerca_r_2(const list<int> &s, int x,
     list<int>::const_iterator it);
/* Pre: it apunta a un elemento de s o a s.end(). x no es
un elemento de la sublista comprendida entre s.begin() y
el elemento anterior al que apunta it. */
/* Post: El resultado indica si x es un elemento de s. */
```

Búsqueda Dicotómica (inmersión por Relajación de Post)

El objetivo es obtener un diseño recursivo de

```
int cerca(const vector<int>& v, int n) {
/* Pre: v.size()>0, v ordenado crecientemente */
/* Post: Si n está en v, el resultado es una posición p
tal que v[p]=n; en otro caso, el resultado es -1. */
  return i_cerca(v, n, 0, int(v.size()) - 1); }
```

Para lograrlo introducimos una función auxiliar recursiva que contiene dos parámetros adicionales que delimitan el fragmento de v donde se realiza la búsqueda (ver cerca_dic_r.cc).

```
int i_cerca_dic(const vector<int>& v, int n,
                int esq, int dre);
/* Pre: v.size()>0, v ordenado crecientemente,
0 ≤ esq ≤ v.size(), -1 ≤ dre < v.size(), y
esq ≤ dre + 1 */
/* Post: Si n está en v[esq...dre], el resultado
es una posición p tal que esq ≤ p ≤ dre y
v[p]=n; en otro caso, el resultado es -1. */
```

Divisores Naturales de n (inmersión por Relajación de Post)

Para realizar un diseño recursivo de la acción

```
void div(int n, stack<int>& p) {
/* Pre: 0 < n, p = P y P está vacía */
/* Post: p contiene los divisores naturales de n
   menores que n en orden descendente. */
  i_div(n, n, p); }
```

introducimos una función auxiliar recursiva con un parámetro adicional `i`, que utilizamos para especificar una postcondición más débil, i.e. `p` *contiene los divisores naturales de n menores que* `i`.

```
void i_div(int n, int i, stack<int>& p);
/* Pre: 0 < i ≤ n, p = P y p está vacía */
/* Post: p contiene los divisores naturales de n
menores que i en orden descendente. */
```

La implementación de esta acción está en `divis.cc`.

Divisores Naturales de n (inmersión por Refuerzo de Pre)

Para realizar un diseño recursivo de la acción

```
void div(int n, stack<int>& p) {
/* Pre: 0 < n, p = P y P está vacía. */
/* Post: p contiene los divisores naturales de n
menores que n en orden descendente. */
   i_div_2(n, 1, p);
}
```

introducimos una función auxiliar recursiva con un parámetro adicional `i`, que utilizamos para especificar una precondición más fuerte, i.e. `p` *contiene los divisores naturales de n menores que* `i`.

```
void i_div_2(int n, int i, stack<int>& p);
/* Pre: 0 < i ≤ n, p = P y P contiene los divisores
naturales de n menores que i en orden descendente. */
/* Post: p contiene los divisores naturales de n
menores que n en orden descendente. */
```

La implementación de esta acción está en `divis_2.cc`.

Funciones Recursivas Lineales Finales

Una función es **recursiva lineal** si cada llamada recursiva genera de manera directa, es decir en la misma activación, una o cero llamadas recursivas. Todas los ejemplos de funciones recursivas de este bloque de transparencias son funciones recursivas lineales.

Una función recursiva es **lineal final** si la última instrucción que se ejecuta es la llamada recursiva y si el resultado de la función en el caso recursivo es el resultado de la llamada recursiva. Las funciones `cerca_r`, `cerca_r_2`, `i_cerca_dic`, `i_div_2` e `i_suma_2` de este bloque de transparencias son funciones recursivas lineales finales.

Una función recursiva lineal final **tiene postcondición constante** si su postcondición y la de su llamada recursiva son iguales. Las funciones `cerca_r_2`, `i_div_2` e `i_suma_2` de este bloque de transparencias son recursivas lineales finales con postcondición constante.

Las dos inmersiones de la función `suma(const vector<int>& v)` son *recursivas lineales*. Pero sólo `i_suma_2` es *recursiva lineal final con postcondición constante*.

Función Recursiva Lineal Final con Postcond Constante

```
Tipus2 f(Tipus1 x)
/* Pre: Q(x), x = X */
/* Post: R(X,s) */
{
   Tipus2 s;
   if (c(x)) s = d(x);
   else s = f(g(x))
   return s;
}
```

a) La condición del caso recursivo es la condición de `while`.

b) Las instrucciones del cuerpo del bucle son las del caso recursivo a excepción de la llamada recursiva.

c) Las instrucciones del caso sencillo se aplican una vez que ha terminado la iteración.

Función Recursiva Lineal Final con Postcond Constante

```
Tipus2 f_rec(Tipus1 x)
/* Pre: Q(x), x = X */
/* Post: R(X,s) */
{
   Tipus2 s;
   if (c(x)) s = d(x);
   else s = f_rec(g(x))
   return s;
}
Tipus2 f_iter(Tipus1 x)
/* Pre: Q(x), x = X */
/* Post: R(X,s) */
{
   while (not c(x)) x = g(x);
   Tipus2 s = d(x);
   return s;
}
```

Transformar i_suma_2(v,i,s_i) $\Rightarrow$ suma_iter(v,i,s_i)

```
int i_suma_2(const vector<int>& v, int i, int s_i) {
/* Pre: 0 ≤ i ≤ v.size(),
   s_i es la suma de los elementos de v[0...i-1]*/
/* Post: el resultado es la suma de los
   elementos de v */

  int s;
  if (i == v.size()) s = s_i;
  else {
    s = i_suma_2(v, i+1, s_i + v[i]);
    /* HI: s es la suma de los elementos de v */
  }
  return s;
}
```

Transformar i_suma_2(v,i,s_i) $\Rightarrow$ suma_iter(v,i,s_i)

La precondición se cumple antes y después de cada iteración, luego sirve como invariante. La demostración de terminación es la misma.

```
int suma_iter(const vector<int>& v, int i, int s_i) {
/* Pre: 0 ≤ i ≤ v.size(),
   s_i es la suma de los elementos de v[0...i-1]*/
/* Post: el resultado es la suma de los elementos de v */

  int s;
  while (i ≠ v.size()) {
     s_i = s_i + v[i];
     i = i + 1;
  }
  s = s_i;
  return s;
}
```

Transformar suma_iter(v,i,s_i) $\Rightarrow$ suma_iter(v)

Convertimos parámetros de inmersión en variables locales inicializadas con los argumentos de la llamada `i_suma_2(v,0,0)` en `suma(v)`. Eliminamos la asignación `s = s_i`, utilizando `s` en lugar de `s_i`.

```
int suma_iter(const vector<int>& v) {
/* Pre: cierto */
/* Post: el resultado es la suma de los elementos de v */
  int s = 0;
  int i = 0;
  /* Inv: 0 ≤ i ≤ v.size(),
     s es la suma de v[0...i-1]*/
  while (i ≠ v.size()) {
     s = s + v[i];
     i = i + 1;
  }
  return s;
}
```