Corrección de programas recursivos

Primero formalizaremos los efectos de las llamadas a funciones en general.

Después les añadiremos los elementos específicos de las funciones recursivas.

Corrección de programas recursivos

Consideramos funciones de la siguiente forma

```
T2 f(T1 x)
/* Pre: Q(x) */
/* Post R(x,resultado) */
```

Si tenemos *acciones*, la post habrá de distinguir entre el estado inicial y el estado final de los parámetros por referencia

Corrección de programas recursivos

El efecto de una llamada

`{A} m=f(n) {?}`

se define así:

"Después de la llamada se cumple **`R(n,m)`** si antes se cumple la precondición (**`A=>Q(n)`**) y **`n`** es evaluable sin error"

Corrección de programas recursivos

```
int potencia (int a, int b)

/* Pre: a>=0, b>0 */

/* Post: el resultado es a^b (a
multiplicado por sí mismo b veces) */
```

Después de

```
x=potencia(2*y,z+1);
```

si **2y>=0** y **z+1>0** se cumple que **$x=(2y)^{z+1}$**

Corrección de programas recursivos

Una función recursiva contiene al menos dos casos, uno sin recursividad y otro con ella

```
T2 f(T1 x)
/* Pre: Q(x) */
/* Post R(x,resultado) */
```

```
T2 s;
if (CD(x)) s=INR1(x);
else {
  r=f(INR2(x));
  /* HI: R (INR2(x),r) */
  s=INR3(x,r);
}
return s;
```

INRi: instrucciones no recursivas

Corrección de programas recursivos

La función es correcta si lo es cada uno de sus casos y si sus parámetros decrecen en cada llamada:

Existe una función t sobre los parámetros de f tal que:

- **`not`** `CD(x)` => t es un natural
- t decrece estrictamente para cada llamada:

`not` `CD(x)` => t(**`INR2(x)`**) < t(**`x`**)

Corrección de programas recursivos

La corrección se basa en el principio de inducción:

- la **base de inducción** son los casos directos
- el **paso de inducción** son los casos recursivos
- la **hipótesis de inducción** es suponer que los resultados de las llamadas son correctos si sus parámetros son menores que los originales

Ejemplos de justificación de programas recursivos

- producto rápido de números enteros
- pilas iguales
- buscar en una cola
- tamaño de un árbol binario
- sumar k a todos los elementos de un árbol binario
- mínimo de un árbol binario (ojo!)

Diseño recursivo por inmersión

Una inmersión de una función `f` es otra función `g` más general (tiene más datos o calcula más cosas, pero permite indirectamente calcular la propia `f`).

A veces `f` no es programable recursivamente pero se puede encontrar una inmersión `g` que sí lo sea.

Puede hacerse añadiendo datos, resultados o los dos a la vez.

Diseño recursivo por inmersión

Inmersión por *datos*:

Función original: `T1 f(T2 x);`

Inmersión: `T1 g(T2 x, T3 y);`

`g` es inmersión de **`f`** si para todo x existe una expresión E de forma que `f(x) = g(x,E)`

Diseño recursivo por inmersión

Ejemplo: buscar un elemento en un vector; la inmersión solo busca *entre dos posiciones determinadas*

```
int busqueda (const vector<int>& v, int x)
// busca x en todo v
int busqueda_i(const vector<int>& v, x, i, j)
// busca x en v[i..j]
busqueda(v, x) = busqueda_i(v, x, 0, v.size-1)
```

Diseño recursivo por inmersión

Inmersión por *resultados*:

Función original: `T1 f(T2 x);`

Inmersión: `pair <T1, T3> g(T2 x);`

`g` es inmersión de **`f`** si para todo `x: f(x) = g(x).first`

Diseño recursivo por inmersión

Ejemplo: función de Fibonacci

```
fib(0) = 0; fib(1) = 1;
n>1: fib(n) = fib(n-1) + fib(n-2)

int fibonacci(int n) // calcula fib(n)
```

Inmersión: calcula dos `fib` consecutivos

```
pair <int, int> fibonacci_i(int n)
// calcula fib(n) y fib(n-1)
```

Ejemplos de programas recursivos por inmersión vistos en PRO1

(los repasaremos en unos momentos; consultar también documento “Algorismes Fonamentals” de PRO1)

- búsqueda dicotómica (vectores)
- mergesort (vectores)

Inmersiones por debilitamiento de la postcondición

La inmersión se obtiene a partir de la original, con una postcondición más débil (hay que calcular "menos")

- sumar los elementos de un vector
- búsqueda dicotómica (apuntes, págs. 27-29)
- mergesort

Suma de un vector por debilitamiento de la postcondición

```
int suma_vect_int(const vector<int> &v)
/* Pre: v.size()>0 */
/* Post: el valor retornat és la suma de v */
```

Inmersión:

```
int i_suma_vect_int(const vector<int> &v, int i)
/* Pre: 0<=i<v.size() */
/* Post: el valor retornat és la suma de v[0..i] */
```

Suma de un vector por debilitamiento de la postcondición

Si implementamos correctamente **`i_suma_vect_int`**, la función original queda

```
int suma_vect_int(const vector<int> &v)
/* Pre: v.size()>0 */
/* Post: el valor retornat és la suma de tots els elements del vector */
return i_suma_vect_int(v,v.size()-1);
```

`v.size()-1` cumple la Pre de `i_suma_vect_int`:

`0<= v.size()-1 <v.size()`

Ejemplos de inmersiones por refuerzo de la precondición

La inmersión se obtiene a partir de la original, con una precondición más fuerte (entra “más” ya calculado)

- sumar los elementos de un vector

Suma de un vector por refuerzo de la precondición

```
int suma_vect_int(const vector<int> &v)
/* Pre: v.size()>0 */
/* Post: el valor retornat és la suma de v */
```

Inmersión:

```
int ii_suma_vect_int(const vector<int> &v, int i,
int s)
/* Pre: 0<=i<v.size(), s és la suma de v[0..i] */
/* Post: el valor retornat és la suma de v */
```

Suma de un vector por refuerzo de la precondición

Si implementamos correctamente **`ii_suma_vect_int`**, la función original queda

```
int suma_vect_int(const vector<int> &v)
/* Pre: v.size()>0 */
/* Post: el valor retornat és la suma de v */
return ii_suma_vect_int(v,0,v[0]);
```

`0, v[0]` cumplen la Pre de `ii_suma_vect_int`:

`0<= 0 <v.size(), v[0] es la suma de v[0..0]`

Relación entre recursividad lineal final e iteración

• rec. final: la llamada es la última instrucción de cada rama

• rec. lineal: solo una llamada recursiva en cada rama

• la transformación de una operación recursiva lineal final a iterativa es automática **(**pág. 33 de los apuntes)

Relación entre recursividad lineal final e iteración

Inmersiones para conseguir recursividad final

- versión 2 del mínimo de un árbol (el resultado pasa a ser un parámetro por referencia)
- nueva versión de producto rápido