Diseño de algoritmos recursivos

Gonzalo Méndez es el autor principal de este tema

Facultad de Informática - UCM

5 de noviembre de 2013

Introducción

Solución recursiva de un problema:

- Se conoce la solución del problema para un conjunto de datos *simple*. (Caso base o directo)
- Se conoce como resolver el problema para el resto de los datos a partir de la solución del problema para casos más sencillos.(Caso recursivo)

Esquema general de un procedimiento recursivo

```
void nomPro(τ1x1 , ... , τnxn , δ1 & y1 , ... , δm & ym )
{
// declaración de constantes y variables locales
   τ1x11 ; ... ; τnx1n ; ... ; τ1xk1 ; ... ; τnxkn ;
   δ1y11 ; ... ; δmy1m ; ... ; δ1yk1 ; ... ; δmykm ;

   if ( d(x⃗) )
      y⃗ = g(x⃗);
   else if ( ¬ d(x⃗) ) {
      x⃗1 = s1( x⃗ );
      nombreProc(x⃗1 , y⃗1 );
      ...
      x⃗k = sk( x⃗ );
      nombreProc(x⃗k , y⃗k );
      y⃗ = c(x⃗, y⃗1 , ... , y⃗k );
   }
}
```

Tipos de recursión

Recursión simple (o lineal): cada caso recursivo realiza exactamente una llamada recursiva.

$\{n \geq 0\}$
```
fun factorial (int n) return int r
```
$\{r = n!\}$

```
int factorial ( int n ){
  int r;
  if      ( n == 0 ) r = 1;
  else  r = n * factorial(n-1);  // (n > 0)
  return r;
}
```

1. Condición del caso directo: $d(n) = (n == 0)$
2. Función de cálculo del caso directo: $g(n) = 1$
3. Función sucesor: $s(n) = n - 1$
4. Función de combinación: $c(n, fact(s(n))) = n * fact(s(n))$

- Se pueden tener varios casos directos o varias descomposiciones para el caso recursivo. Las alternativas deben ser **exhaustivas** y **excluyentes**.
- Ejemplo: multiplicación de dos naturales por el método del *campesino egipcio*.

$\{(a \geq 0) \wedge (b \geq 0)\}$
fun prod (int a, int b) return int r
$\{r = a * b\}$

```
int prod ( int a, int b )
{
  int r;

  if      ( b == 0 )                    r = 0;
  else if ( b == 1 )                    r = a;
  else if ( b > 1 && (b %2 == 0) )      r = prod(2*a, b/2);
  else if ( b > 1 && (b %2 == 1) )
          r = prod(2*a, b/2) + a;
  return r;
}
```

Tipos de recursión.

Recursión final o de cola (*tail recursion*) caso particular de recursión simple donde la función de combinación se limita a transmitir el resultado de la llamada recursiva. El resultado será siempre el obtenido en uno de los casos base.
Ejemplo: cálculo del mcd por el algoritmo de Euclides.

$\{(a > 0) \wedge (b > 0)\}$
`fun mcd (int a, int b) return int r`
$\{r = mcd(a, b)\}$

```
int mcd( int a, int b )
{
  int m;

  if      ( a == b ) m = a;
  else if ( a >  b ) m = mcd(a-b, b);
  else if ( a <  b ) m = mcd(a, b-a);
  return m;
}
```

Tipos de recursión

Recursión múltiple: al menos en un caso recursivo, se realizan varias llamadas recursivas.

$\{n \geq 0\}$
`fun fib (int n) return int r`
$\{r = fib(n)\}$

```
int fib( int n )
{
  int r;

  if      ( n == 0 ) r = 0;
  else if ( n == 1 ) r = 1;
  else if ( n >  1 ) r = fib(n-1) + fib(n-2);

  return r;
}
```

Diseño de algoritmos recursivos

1. **Planteamiento recursivo.** Realizar una descomposición recursiva de la postcondición. Definir la estrategia recursiva para alcanzar la postcondición.
2. **Análisis de casos.** Identificar las condiciones que permiten discriminar los casos directos de los recursivos. Deben tratarse de forma exhaustiva y mutuamente excluyente todos los casos contemplados en la precondición:

$$P_0 \Rightarrow d(\vec{x}) \vee \neg d(\vec{x})$$

3. **Caso directo.** Encontrar la acción A_1 que resuelve el caso directo:

$$\{P_0 \wedge d(\vec{x})\} A_1 \{Q_0\}$$

Si hubiese más de un caso directo, repetiríamos este paso para cada uno de ellos.

4. **Descomposición recursiva.** Obtener la función sucesor $s(\vec{x})$ que nos proporciona los datos que empleamos para realizar la llamada recursiva.
   Si hay más de un caso recursivo, obtenemos la función sucesor para cada uno de ellos.
5. **Función de acotación y terminación.** Determinamos si la función sucesor escogida garantiza la terminación de las llamadas. Definir una función cota que estime el número de llamadas restantes hasta alcanzar un caso base. Justificar que se decrementa en cada llamada.
   Si hay más de un caso recursivo, se ha de garantizar la terminación para cada uno de ellos.

6. **Llamada recursiva.** Cada una de las descomposiciones recursivas ha de permitir realizar la(s) llamada(s) recursiva(s), es decir, la función sucesor debe proporcionar unos datos que cumplan la precondición de la función recursiva.

$$P_0 \wedge \neg d(\vec{x}) \Rightarrow P_0[\vec{x}/s(\vec{x})]$$

7. **Función de combinación.** Obtener la función de combinación, que, en el caso de la recursión simple, será de la forma $\vec{y} = c(\vec{x}, \vec{y}')$.
   Si hubiese más de un caso recursivo, habría que encontrar una función de combinación para cada uno de ellos.
8. **Escritura del caso recursivo.** Eliminar las variables auxiliares que no sean necesarias.

- Ejemplo: una función que dado un natural n calcule la suma de los dobles de los naturales hasta n:

  Especificación

  $\{n \geq 0\}$
  `fun sumaDoble (int n) return int r`
  $\{r = \sum i : 0 \leq i \leq n : 2 * i\}$

```
int sumaDoble ( int n )
{
  int r;

  // cuerpo de la función

  return r;
}
```

1. Planteamiento recursivo. Descomposición de la postcondición:

$$\sum i : 0 \leq i \leq n : 2 * i = (\sum i : 0 \leq i \leq n - 1 : 2 * i) + 2 * n$$

Estrategia recursiva:

$$sumaDoble(n) = sumaDoble(n - 1) + 2 * n$$

2. Análisis de casos

Solución directa: $n = 0$, donde el resultado es 0.

Distinción de casos:

$$d(n) : n = 0$$

$$\neg d(n) : n > 0$$

Cubre exhaustivamente todos los casos posibles

$$n \geq 0 \Rightarrow n = 0 \vee n > 0$$

3. Solución en el caso directo.

$$\{\ \texttt{n = 0}\ \}$$
$$A_1$$
$$\{\ r = \sum i : 0 \leq i \leq n : 2 * i\ \}$$

Se resuelve trivialmente con la asignación

$$A_1 \equiv r = 0$$

4. Descomposición recursiva. Cálculo de la función sucesor.

$$s(n) = n - 1$$

5. Función de acotación y terminación. El tamaño del problema viene dado por el valor de n con $n \geq 0$:

$$t(n) = n$$

Se decrementa en cada llamada recursiva:

$$t(s(n)) < t(n) \Leftrightarrow n - 1 < n$$

6. Es posible hacer la llamada recursiva. En el caso recursivo se cumple

$$P_0(n) \wedge \neg d(n) \Rightarrow P_0(s(n))$$

$$n \geq 0 \wedge n > 0 \Leftrightarrow n > 0 \Rightarrow n - 1 \geq 0$$

Como se ve, se trata de, a partir de la precondición y la condición del caso recursivo, obtener la precondición expresada en función del sucesor.

7. **Función de combinación.**
   La postcondición se alcanza con una asignación, si r' es el resultado devuelto por la llamada recursiva:

$$r = r' + 2 * n$$

8. **Escritura del caso recursivo.**
   Siguiendo el esquema teórico, tenemos que el caso recursivo se escribe:

```
{ P0 ∧ n > 0 }
  n' = n-1;
  r' = sumaDoble(n');
  r  = r' + 2*n;
{ Q0 }
```

9. Podemos evitar el uso de las dos variables auxiliares:

```
r = sumaDoble(n-1) + 2*n;
```

La función obtenida es:

```
int sumaDoble ( int n ) {
// Pre: n ≥ 0
  int  r;

  if      ( n == 0 ) r = 0;
  else if ( n >  0 ) r = sumaDoble( n-1 ) + 2*n;
  return r;
// Post: r = ∑ i : 0 ≤ i ≤ n : 2 * i
}
```

- Ejemplo: suma de las componentes de un vector de enteros.

  Especificación

  $\{longitud(v) \geq num\}$

  `fun sumaVec (int v[], int num) return int r`

  $\{r = \sum i : 0 \leq i < num : v[i]\}$

```
int sumaVec ( int v[], int num )
{
  int  r;
  // cuerpo de la función
  return r;
}
```

1. Planteamiento recursivo. Descomposición de la postcondición:
$\sum i : 0 \leq i < num : v[i] = \sum i : 0 \leq i < num-1 : v[i]+v[num-1]$

   Estrategia recursiva:

$$sumaVec(v, num) = sumaVec(v, num - 1) + v[num - 1]$$

2. Análisis de casos.
   Solución directa: $num = 0$, pdonde el resultado es 0.
   Distinción de casos:

$$d(v, num) : num = 0$$

$$\neg d(v, num) : num > 0$$

¿Qué ocurre si $num < 0$?
Podemos optar por:

- exigir en la precondición que sólo son válidas invocaciones donde $num \geq 0$, o
- tratar también el caso en el que $num < 0$ devolviendo también 0

Optando por la segunda solución: $d(v, num) : num \leq 0$

$$P_0 \Rightarrow (num < 0) \vee (num = 0) \vee (num > 0) \Leftrightarrow true$$

3. Solución en el caso directo.

$$\{ \ P_0 \wedge num \leq 0 \ \}$$
$$A_1$$
$$\{ \ r = \sum i : 0 \leq i < num : v[i] \ \}$$

$A_1 \equiv r = 0;$

4. Descomposición recursiva.

$$s(v, num) = (v, num - 1)$$

5. Función de acotación y terminación.

Al avanzar la recursión, *num* se va acercando a 0

$$t(v, num) = num$$

Se cumple

$$num - 1 < num$$

6. Llamada recursiva.
   $longitud(v) \geq num \wedge num > 0 \Rightarrow longitud(v) \geq num - 1$
7. Función de combinación.
   Se resuelve con la asignación

$$r = r' + v[num - 1];$$

8. Escritura del caso recursivo.
   Bastará con la asignación

```
r = sumaVec(v, num-1) + v[num-1];
```

La función obtenida es:

```
//fun sumaVec (int v[], int num) return int r
int sumaVec ( int v[], int num ) {
// Pre: longitud(v) ≥ num
  int  r;

  if ( num <= 0 )
     r = 0;
  else if ( num >  0 )
     r = sumaVec(v, num-1) + v[num-1];

  return r;
// Post: r = ∑ i : 0 ≤ i < num : v[i]
}
```