Eliminación de la recursión y transformación de programas

Fernando Schapachnik[1]

[1]Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Algoritmos y Estructuras de Datos II,
primer cuatrimestre de 2015

(2) Revisemos la recursión

- Empecemos por un clásico de la Alta Edad Media: la sucesión de Fibonacci.
- $Fib(0) = 0$
- $Fib(1) = 1$
- $Fib(n+2) = Fib(n+1) + Fib(n)$
- Miremos $Fib(5) = Fib(4) + Fib(3)$.
- Pero más en detalle: expandamos $Fib(4)$.
- $Fib(5) = (Fib(3) + Fib(2)) + Fib(3)$
- ¡Ajá! Estamos calculando $Fib(3)$ dos veces.
- (En realidad, ya vemos en la definición que calculamos $Fib(n)$ dos veces.)
- ¿No podemos hacer algo más piola?

(3) Arreglando Fibonacci

- Idea:
  - Cada vez que calculo $Fib(m)$ tengo que calcular $Fib(m-1)$ y $Fib(m-2)$.
  - Además, $Fib(m+1)$ va a necesitar a $Fib((m+1)-1)$ y a $Fib((m+1)-2)$, es decir, $Fib(m-1)$, que ya lo calculé recién.
  - ¿Y si hago que $Fib(m)$ me devuelva también al $Fib(m-1)$ que utilizó?
- La Fibonacci arreglada sería así:
- $Fib'(n+1) = \langle \Pi_1(Fib'(n)) + \Pi_2(Fib'(n)), \Pi_1(Fib'(n)) \rangle$
- $Fib(n) = \Pi_1(Fib'(n))$
- $Fib'(0) = \langle 0, 0 \rangle$
- $Fib'(1) = \langle 1, 0 \rangle$
- La versión original toma $O(2^n)$ y mientras que la nueva es lineal en n.

(4) Inmersión de rango

⚠ Eso que hicimos se llama inmersión de rango, y consiste en agregar un nuevo parámetro de salida a la función.

- La idea de este nuevo parámetro es darnos "algo más" que lo estrictamente necesario. Ese "algo más" nos sirve para calcular más eficientemente otro resultado.
- A esta técnica también se la conoce como generalización de funciones. En el ejemplo, $Fib'()$ es más general que $Fib()$.

⚠ La *idea* de la inmersión de rango puede servir también para hacer varias cosas "a la vez".

  - Ejercicio: escribir una función que compute la cantidad de nodos y la altura máxima de un árbol binario de naturales.

- No es la única inmersión...

(5) Inmersión de dominio

- La idea de la inmersión de dominio es similar.
- Se agrega un parámetro de entrada, que va conteniendo resultados intermedios.
- Pensemos en Promedio() de una secuencia, que requeriría la suma y la cantidad.
- Podemos hacerlo mejor:
  - Promedio(<>, suma, cantidad) $\equiv$ suma/cantidad
  - Promedio(a $\bullet$ S, suma, cantidad) $\equiv$ Promedio(S, suma+a, cantidad+1)

(6) Otro clásico

- Veamos otro clásico: $x^0 = 1$, $x^{n+1} = x^n.x$
- ¿Complejidad? n multiplicaciones y llamadas recursivas.
- ¿Podemos mejorarlo?
  1. Idea: expandamos x^n
  2. $x^{n+1} = (x^{n-1}.x).x$, pero como la multiplicación asocia:
  3. $x^{n+1} = x^{n-1}.x.x$
  4. Pero $x.x$ es la definición de x^2.
  5. $x^{n+1} = x^{n-1}.x^2$
  6. Y repitiendo esta idea (en el caso $n+1$ par):
  7. $x^{n+1} = (x^2)^{(n+1)/2}$
- Esta función hace aprox. la mitad de multiplicaciones.
- En el paso (2) lo que hicimos fue desplegar la definición de x^n.
- En el (5) hicimos un plegado.

(7) Folding/unfolding

A esta técnica se la conoce como folding/unfolding o plegado/desplegado.

- El desplegado consiste en reemplazar la definición de la función por su expresión.
- El plegado puede pensarse como la inversa de la anterior. Se trata de reemplazar la definición de una expresión por una llamada a la función.

(8) Folding/unfolding (cont.)

⚠ La estrategia más común consiste en desplegado, rearreglo, plegado.

- Ejemplo:
  - F(x, y) = if predicado_caro(x) then A(y) else B(y)
  - En lugar de evaluar F(x, y)+F(x, z), despliego:
  - if predicado_caro(x) then A(y) else B(y) + if predicado_caro(x) then A(z) else B(z)
  - Luego reacomodo: if predicado_caro(x) then A(y)+A(z) else B(y)+B(z)
  - Supongamos que A() y B() distribuyen con la suma: if predicado_caro(x) then A(y+z) else B(y+z)
  - Por último, vuelvo a plegar: F(x, y+z)
- De alguna manera, el refactoring es una forma de plegado.

(9) Folding/unfolding en bases de datos

- Imaginemos una empresa multinacional. Queremos obtener la dirección de un cliente en particular. Supongamos 10 millones de registros de los cuales 8 millones son argentinos.
- En el lenguaje más común para consultas de bases de datos, SQL, eso se haría con `SELECT direccion FROM clientes_argentinos WHERE nombre='Nicolás Rosner'`
- Además, `clientes_argentinos` estaría tradicionalmente definido como
  `SELECT * FROM clientes WHERE pais='Argentina'`
- Hacer las consultas en el orden propuesto implica recorrer 10 millones de registros, seleccionar de alguna manera a los 8 millones argentinos, y luego volverlos a recorrer para quedarme con 1.
- Usemos plegado y desplegado.

(10) Folding/unfolding en bases de datos (cont.)

- Desplegamos `clientes_argentinos`:
  `SELECT direccion FROM (SELECT * FROM clientes WHERE pais='Argentina') WHERE nombre='Nicolás Rosner'`
- Reacomodamos (usando reglas propias de este lenguaje, que en AlgoII no nos importan): `SELECT direccion FROM (SELECT * FROM clientes WHERE pais='Argentina' AND nombre='Nicolás Rosner')`
- Seguimos reacomodando:
  `SELECT direccion FROM clientes WHERE pais='Argentina' AND nombre='Nicolás Rosner'`
- De esta manera resolvemos el problema en una sola pasada.
- Los motores de bases de datos hacen estas cosas de manera automática.

(11) El problema de la recursión

- Las computadoras reales son un problema.
- En las ficticias, los algoritmos recursivos son geniales.
- En las reales,
  - cada llamada a función requiere pasar los parámetros a la pila, que es costoso, y
  - consumen espacio de pila, que es acotado.
- En cambio, los compiladores suelen estar buenos.
- O mejor dicho, hubo gente que se dio cuenta de que algunos de estos problemas se podían solucionar automáticamente.

(12) Eliminación de recursión

- Algunos tipos de funciones recursivas se pueden hacer iterativas mecánicamente.
- Ejemplo:

```
// Precondición: x está en S.
función BuscarPosición(int x, secu S, int pos)
if (x==prim(S))
  return pos
else
  return BuscarPosición(x, fin(S), pos+1)
```

- ¿Cuál es la última operación que se hace?

(13) Eliminación de recursión (cont.)

- El esqueleto sería así:

```
función F(parámetros)
if (caso base(parámetros))
  return algún valor
else
  return F(achicar(parámetros))
```

- La podemos transformar en

```
función F(parámetros)
while !(caso base(parámetros))
  parámetros= achicar(parámetros)
return algún valor
```

- ¿Cómo sería con BuscarPosición()?

(14) Recursión a la cola

⚠ Estas funciones se llaman recursivas a la cola.

- (Con más precisión, se llaman *recursivas lineales a la cola* o *recursivas lineales finales*.)
- Su característica es que su último paso es la llamada recursiva y que ésta es la única llamada recursiva.
- Pueden transformarse automáticamente a iterativas.
- Los compiladores suelen hacerlo.
- Veamos otro caso:

```
función Sumatoria(secu S)
if (vacía?(S))
  return 0
else
  return prim(S)+Sumatoria(fin(s))
```

- ¿Es recursiva a la cola? No.
- ¿Está todo perdido? Tampoco.

(15) Funciones recursivas lineales no a la cola

- También están las recursivas lineales no a la cola.
- Son aquéllas que también tienen una única llamada recusiva (por eso *lineales*), pero donde ésta no es la última operación.
- Veamos de nuevo la Sumatoria:

```
función Sumatoria(secu S)
if (vacía?(S))
  return 0
else
  return prim(S)+Sumatoria(fin(s))
```

- ¿Se acuerdan de la inmersión?

```
función SumatoriaGeneralizada(int acumulador, secu S)
if (vacía?(S))
  return acumulador
else
  return SumatoriaGeneralizada(prim(S)+acumulador, fin(s))
```

- También pueden transformarse automáticamente.

(16) Funciones recursivas no lineales

También hay funciones recursivas no lineales.

- Son aquellas que tienen más de una llamada recursiva.
  - Múltiple: Funciones como *Fib*(), que tienen más de una llamada recursiva, pero no es la última operación.
  - Anidada: Tienen más de una llamada recursiva, anidada. Además, la llamada recursiva es la última operación.
- Anticipo: Tenemos una forma de eliminar la recursión de las funciones anidadas, y también sabemos llevar las múltiples a anidadas.
- ¿Siempre? No, pero en muchos casos.

(17) Recursiones múltiples

- Veamos otro ejemplo:
  - Pre(nil) $\equiv$ <>
  - Pre(bin(i, x, d)) $\equiv$ x $\bullet$ (Pre(i) & Pre(d))
- ¿Cómo la podemos llevar a anidada? Agregando un parámetro de acumulación (en particular, una secuencia, que representa el resultado parcial).
- Buscamos que Pre(ab) $\equiv$ PreA(<>, ab) (es decir, que al comenzar el acumulador esté vacío), y
- PreA(<>, ab) $\equiv$ PreA(S, nil) (para alguna secuencia S, es decir, que tenga el resultado en el acumulador al terminar de recorrer el árbol).

(18) Recursiones múltiples (cont.)

- Veamos cómo quedaría:
  1. Pre(ab) $\equiv$ PreA(<>, ab)
  2. PreA(acum, nil) $\equiv$ acum
  3. PreA(acum, bin(i, x, d)) $\equiv$ PreA(acum & (x $\bullet$ PreA(<>, i)), d)
  4. (auxiliar) S & Pre(ab) $\equiv$ S & PreA(<>, ab) $\equiv$ PreA(S, ab)
- ¿Está bien? Pongamos un árbol en (1) y apliquemos plegado y desplegado.
  - Pre(bin(i, x, d)) $\equiv$ PreA(<>, bin(i, x, d))
  - (desplegando por (3)) $\equiv$ PreA(x $\bullet$ PreA(<>, i), d)
  - (plegando por (1)) $\equiv$ PreA(x $\bullet$ Pre(i), d)
  - (usando (4)) x $\bullet$ Pre(i) & Pre(d)

(19) Recursiones múltiples (cont.)

- PreA(acum, bin(i, x, d)) $\equiv$ PreA(acum & (x $\bullet$ PreA($<>$, i)), d)
- Lo que nos quedó es una recusión anidada, que parece más complicada. Por suerte, tenemos herramientas para eliminarla a partir de ahí.

⚠ Notemos que lo que hicimos fue proponer la transformación y luego demostrar que es correcta.

- Vamos a ver ahora cómo eliminar la recursión de las funciones recursivas anidadas.

(20) Funciones recursivas anidadas

- En estos casos la mejor estrategia suele ser el uso de una pila.
- Vamos a querer que PreG(p, acum) $\equiv$ PreA(<>, ab) (para alguna pila p)
- ¿Para qué serviría la pila?
- Propongo:
- PreG(apilar(ab, p), acum) $\equiv$ PreG(p, PreA(acum, ab))
- Tiene sentido, porque si p es la pila vacía y acum la secuencia vacía, nos queda: PreG(apilar(ab, Vacía), <>) $\equiv$ PreG(Vacía, PreA(<>, ab)), que ya sabemos que es igual a Pre(ab).

⚠ Ahora, en lugar de inventar una transformación y luego probar que es correcta, vamos a deducirla.

(21) Funciones recursivas anidadas (cont.)

- Al igual que antes, pongamos un árbol en la pila y veamos qué sucede.
  1. PreG(apilar(bin(i, x, d), p), acum) ≡
     PreG(p, PreA(acum, bin(i, x, d)))
  2. (despliego PreA())
     ≡ PreG(p, PreA(acum & (x • PreA(<>, i)), d))
  3. (pliego por (1))
     ≡ PreG(apilar(d, p), acum & (x • PreA(<>, i)))
  4. (reacomodo)
     ≡ PreG(apilar(d, p), (acum • x) & PreA(<>, i)))
  5. (uso el lema)
     ≡ PreG(apilar(d, p), PreA((acum • x), i)))
  6. (pliego por (1))
     ≡ PreG(apilar(i, apilar(d, p)), acum • x)

(22) Funciones recursivas anidadas (cont.)

- Nos queda:
  1. Pre(ab) $\equiv$ PreG(apilar(ab, Vacía), <>)
  2. PreG(Vacía, acum) $\equiv$ acum
  3. PreG(apilar(nil, p), acum) $\equiv$ PreG(p, acum)
  4. PreG(apilar(bin(i, x, d), p), acum) $\equiv$
     PreG(apilar(i, apilar(d, p)), acum $\bullet$ x)
- ¿Qué tipo de función es? Recursiva a la cola. Entonces podemos aplicar lo que ya sabemos.

(23) Otras técnicas de transformación de programas

- Generalización de género. Ejemplo: tal vez me conviene trabajar con enteros en lugar de naturales, si tengo que restar.
- Si voy a usar el resultado de una función más de una vez, probablemente convenga tenerlo en una variable.
- De manera más general, si ya computé resultados caros, puede tener sentido almacenarlos (en memoria o en disco) en lugar de recomputarlos.
- Precomputar valores. Por ejemplo, si mi programa va a intentar factorizar, tal vez me convenga tener almacenados los n primeros números primos.

(24) Otras técnicas de transformación (cont.)

- Inversión del orden de cómputo.
- Evaluación parcial. Especializar en valores particulares. Ejemplo:

```
función potencia(nat a, nat b)
 if b<=4 then potencia'(a, b)
 else a.potencia(a, b-1)

función potencia'(nat a, nat b<=4)
 if b==0 then 1
 elsif b==1 then a
 else
  c= a.a
  if b==2 then c elsif b==3 then c.a else c.c
```

- Conceptos relacionados: curryficación, especialización.

(25) Tarea

- Hacer un programa que genere árboles binarios al azar.
- Programar Pre(), PreA(), PreG() y una versión iterativa.
- ¡Tomar tiempos! (comando `time` en Unix)

(26) Repaso

Vimos:

- Inmersiones de rango y de dominio, que eran tipos específicos de generalización de funciones, y que permitían obtener versiones más eficientes.
  - A veces reduciendo la complejidad,
  - a veces, sólo las constantes,
  - pero siempre preservando la corrección.
- Una técnica de transformación muy útil: plegado y desplegado.
- Los problemas de la recursión.
- Cómo eliminarla
  - de las funciones recursivas a la cola y
  - de las funciones recursivas lineales no a la cola.
- Y para las funciones recursivas no lineales,
  - cómo transformar algunas recursiones múltiples en anidadas, y
  - cómo eliminar la recursión de algunas recursiones anidadas.
- Cuando había que obtener una generalización, a veces la deducimos, otras la proponemos y luego demostramos.
- Y de yapa, técnicas varias de transformación.

(27) Bibliografía

- "Specification and Transformation of Programs", Helmut A. Partsch, Springer-Verlag, 1990.
- "Programación Metódica", José Luis Balcázar, Mc Graw Hill, 1993.