

Eliminación de la recursión y transformación de programas

Fernando Schapachnik¹

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

Algoritmos y Estructuras de Datos II,
primer cuatrimestre de 2011

(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 el $Fib(m - 1)$ que utilizó?
- La Fibonacci arreglada sería así:
 - $Fib'(0) = \langle 0, 0 \rangle$
 - $Fib'(1) = \langle 1, 0 \rangle$
 - $Fib'(n + 1) = \langle \Pi_1(Fib'(n)) + \Pi_2(Fib'(n)), \Pi_1(Fib'(n)) \rangle$
 - $Fib(n) = \Pi_1(Fib'(n))$
- La versión original toma $O(2^n)$ mientras que la nueva es $O(n)$.

(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:
 - $\text{Promedio}(<>, \text{suma}, \text{cantidad}) \equiv \text{suma}/\text{cantidad}$
 - $\text{Promedio}(a \bullet S, \text{suma}, \text{cantidad}) \equiv \text{Promedio}(S, \text{suma}+a, \text{cantidad}+1)$

(6) Otro clásico

- Veamos otro clásico: $x^0 = 1$, $x^{n+1} = x^n \cdot x$
- ¿Complejidad? n multiplicaciones y llamadas recursivas.
- ¿Podemos mejorarlo?
 - 1 Idea: expandamos x^n
 - 2 $x^{n+1} = (x^{n-1} \cdot x) \cdot x$, pero como la multiplicación asocia:
 - 3 $x^{n+1} = x^{n-1} \cdot x \cdot x$
 - 4 Pero $x \cdot x$ es la definición de x^2 .
 - 5 $x^{n+1} = x^{n-1} \cdot x^2$
 - 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:
 - $F(x, y) = \text{if } \text{predicado_caro}(x) \text{ then } A(y) \text{ else } B(y)$
 - En lugar de evaluar $F(x, y) + F(x, z)$, despliego:
 - $\text{if } \text{predicado_caro}(x) \text{ then } A(y) \text{ else } B(y) + \text{if } \text{predicado_caro}(x) \text{ then } A(z) \text{ else } B(z)$
 - Luego reacomodo: $\text{if } \text{predicado_caro}(x) \text{ then } A(y) + A(z) \text{ else } B(y) + B(z)$
 - Supongamos que $A()$ y $B()$ distribuyen con la suma: $\text{if } \text{predicado_caro}(x) \text{ then } A(y+z) \text{ else } B(y+z)$
 - 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='Pablo Factorovich'`
- 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='Pablo Factorovich'`
- Reacomodamos (usando reglas propias de este lenguaje, que en Algol no nos importan):
`SELECT direccion FROM (SELECT * FROM clientes WHERE pais='Argentina' AND nombre='Pablo Factorovich')`
- Seguimos reacomodando: `SELECT direccion FROM clientes WHERE pais='Argentina' AND nombre='Pablo Factorovich'`
- 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, algún valor)
if (caso base(parámetros))
    return algún valor
else
    return F(achicar(parámetros), f(algún valor))
```

- La podemos transformar en

```
función F(parámetros)
while !(caso base(parámetros))
    algún valor = f(algún valor)
    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 recursiva (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:
 - $\text{Pre}(\text{nil}) \equiv \langle \rangle$
 - $\text{Pre}(\text{bin}(i, x, d)) \equiv x \bullet (\text{Pre}(i) \ \& \ \text{Pre}(d))$
- ¿Cómo la podemos llevar a anidada? Agregando un parámetro de acumulación (en particular, una secuencia, que representa el resultado parcial).
- Buscamos que $\text{Pre}(ab) \equiv \text{PreA}(\langle \rangle, ab)$ (es decir, que al comenzar el acumulador esté vacío), y
- $\text{PreA}(\langle \rangle, ab) \equiv \text{PreA}(S, \text{nil})$ (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 $\text{Pre}(\text{ab}) \equiv \text{PreA}(<>, \text{ab})$
 - ② $\text{PreA}(\text{acum}, \text{nil}) \equiv \text{acum}$
 - ③ $\text{PreA}(\text{acum}, \text{bin}(i, x, d)) \equiv \text{PreA}(\text{acum} \ \& \ (x \bullet \text{PreA}(<>, i)), d)$
 - ④ $(\text{auxiliar } S \ \& \ \text{Pre}(\text{ab}) \equiv S \ \& \ \text{PreA}(<>, \text{ab}) \equiv \text{PreA}(S, \text{ab})$
- ¿Está bien? Pongamos un árbol en (1) y apliquemos plegado y desplegado.
 - $\text{Pre}(\text{bin}(i, x, d)) \equiv \text{PreA}(<>, \text{bin}(i, x, d))$
 - (desplegando por (3)) $\equiv \text{PreA}(x \bullet \text{PreA}(<>, i), d)$
 - (plegando por (1)) $\equiv \text{PreA}(x \bullet \text{Pre}(i), d)$
 - (usando (4)) $x \bullet \text{Pre}(i) \ \& \ \text{Pre}(d)$


(19) Recursiones múltiples (cont.)

- $\text{PreA}(\text{acum}, \text{bin}(i, x, d)) \equiv \text{PreA}(\text{acum} \ \& \ (x \bullet \text{PreA}(<>, i)), d)$
- Lo que nos quedó es una recursión 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 $\text{PreG}(p, \text{acum}) \equiv \text{PreA}(<>, ab)$ (para alguna pila p)
 - ¿Para qué serviría la pila?
 - Propongo:
 - $\text{PreG}(\text{apilar}(ab, p), \text{acum}) \equiv \text{PreG}(p, \text{PreA}(\text{acum}, ab))$
 - Tiene sentido, porque si p es la pila vacía y acum la secuencia vacía, nos queda: $\text{PreG}(\text{apilar}(ab, \text{Vacía}), <>) \equiv \text{PreG}(\text{Vacía}, \text{PreA}(<>, ab))$, que ya sabemos que es igual a $\text{Pre}(ab)$.
-  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.
 - $\text{PreG}(\text{apilar}(\text{bin}(i, x, d), p), \text{acum}) \equiv \text{PreG}(p, \text{PreA}(\text{acum}, \text{bin}(i, x, d)))$
 - (despliegue $\text{PreA}()$)
 $\equiv \text{PreG}(p, \text{PreA}(\text{acum} \ \& \ (x \bullet \text{PreA}(<>, i)), d))$
 - (pliegue por (1))
 $\equiv \text{PreG}(\text{apilar}(d, p), \text{acum} \ \& \ (x \bullet \text{PreA}(<>, i)))$
 - (reacomodo)
 $\equiv \text{PreG}(\text{apilar}(d, p), (\text{acum} \bullet x) \ \& \ \text{PreA}(<>, i)))$
 - (uso el lema)
 $\equiv \text{PreG}(\text{apilar}(d, p), \text{PreA}((\text{acum} \bullet x), i)))$
 - (pliegue por (1))
 $\equiv \text{PreG}(\text{apilar}(i, \text{apilar}(d, p)), \text{acum} \bullet x)$

(22) Funciones recursivas anidadas (cont.)

- Nos queda:
 - 1 $\text{PreG}(\text{Vacía}, \text{acum}) \equiv \text{acum}$
 - 2 $\text{PreG}(\text{apilar}(\text{nil}, p), \text{acum}) \equiv \text{PreG}(p, \text{acum})$
 - 3 $\text{PreG}(\text{apilar}(\text{bin}(i, x, d), p), \text{acum}) \equiv$
 $\text{PreG}(\text{apilar}(i, \text{apilar}(d, p)), \text{acum} \bullet x)$
- ¿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.

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