

### 1.1. Inmersión de Rango / Generalización de Funciones

$$b_n = f(b_{n-1}, b_{n-2}, \dots, b_{n-k})$$
$$c_n = \langle f(\pi_1(c_{n-1}), \pi_2(c_{n-1}), \dots, \pi_k(c_{n-1})), \pi_1(c_{n-1}), \pi_2(c_{n-1}), \dots, \pi_{k-1}(c_{n-1}) \rangle$$

A recursion tree for calculating the 6th Fibonacci number, F(6). The root node is F(6), which branches into F(5) and F(4). F(5) branches into F(4) and F(3). F(4) branches into F(3) and F(2). F(3) branches into F(2) and F(1). The tree shows the sequence of recursive calls and returns, with the final result being F(6) = 8.

## 1.2. Inmersión de Dominio

1

### 1.3. Folding/Unfolding

En folding/unfolding, intentamos obtener una recursión más veloz a través de reducir un poco la expresión original de nuestra función utilizando manipulación simbólica. La derivación del algoritmo de exponenciación rápida sobre monoides es un caso canónico.

### 1.4. Memoization

En el caso de recursiones que podrían ser llamadas muchas veces, son relativamente caras de llamar, y no dependen del estado externo de alguna estructura, una técnica muy común es la utilización de una caché para evitar recomputar datos. Un ejemplo factible es, por ejemplo, si sabemos que se va a computar Fibonacci muchas veces, podemos utilizar un arreglo para guardar los resultados parciales, en cuyo caso lo único que tendríamos que hacer es buscar si el valor ya fue computado, y si no lo fue computarlo y guardarlo en el arreglo.

## 2. Eliminación de la recursión

### 2.1. Pros y contras de la recursión

La utilización de recursión, si bien facilita ampliamente la comprensión de muchos algoritmos que tal vez son más difíciles de comprender en sus versiones recursivas (como el cómputo de altura de un árbol, búsqueda binaria sobre un arreglo, etcétera) y facilita la lectura del código, así como el proceso de eliminación de bugs, tiene varias desventajas respecto de sus contrapartes iterativas en la práctica, fundamentalmente por la forma en la que están diseñadas las computadoras.

En primer lugar, uno de los mayores problemas de la recursión es que cada llamada a una función requiere utilizar espacio en el stack para poner los argumentos de la misma. El espacio de stack acotado, mucho más acotado que la memoria de la máquina, por lo que si la recursión es muy profunda (es decir, el árbol de recursión tiene muchos nodos) el programa podría incurrir en un stack overflow, provocando que el sistema operativo mate el proceso.

En segundo lugar, perdemos la noción de estado implícita. Nótese que si uno necesitase mantener información extra a lo largo de la recursión (como un contador), las alternativas para poder utilizarlo involucran hacer inmersión de rango o de dominio. En los lenguajes imperativos, otra alternativa es utilizar una variable global (de cualquier forma, la utilización de variables globales es una práctica que suele evitarse por motivos de diseño).

En último lugar, suele ser difícil frenar una función recursiva por la mitad en caso de errores. Usualmente hacer esto requiere un tipo de mecanismo más o menos sofisticado para poder avisar a los niveles superiores del árbol de recursión que hubo un error más abajo (véase Maybe en Haskell, y excepciones más en general).

### 2.2. Eliminación de la recursión

Si bien podemos expresar las mismas funciones utilizando recursión que utilizando iteración, hay ciertos tipos de algoritmos recursivos que permiten ser

convertidos a sus contrapartes iterativas automáticamente. En principio, dada cualquier función recursiva uno podría simular el stack con una estructura propia en el lenguaje de programación a elección, y así convertir las llamadas usuales a una función recursiva en una operación sobre nuestra estructura. Si bien esto es una posibilidad, es una forma demasiado complicada de acercarse al problema, y que probablemente no funcione bien.

Analizando casos particulares, las funciones recursivas lineales finales/a cola, es decir, aquellas con una única llamada recursiva que ocurre como último paso de la función, pueden ser automáticamente modificadas para ser iterativas. Por ejemplo, una función recursiva lineal final podría ser similar a:

---

**Algorithm 1** Esqueleto y conversión de una función recursiva lineal final

---

```

1: procedure FUNCIÓN(Parámetros)
2:   if CasoBase(Parámetros) then
3:     return Valor
4:   else
5:     return Función(Achicar(Parámetros))
6:   end if
7: end procedure
8:
9: procedure CONVERTIDA(Parámetros)
10:  while !CasoBase(Parámetros) do
11:    Parámetros  $\leftarrow$  Achicar(Parámetros)
12:  end while
13:  ...
14:  return ValorFinal
15: end procedure

```

---

Otro caso particular son las funciones recursivas lineales que no son finales, es decir, con una única llamada recursiva pero no necesariamente con esta como último paso de la función. En estos casos, utilizando la técnica de inmersión de dominio podemos convertir la recursión en final automáticamente. El último caso en el cual podemos hacer eliminación de la recursión automáticamente es en el caso de las funciones recursivas no lineales múltiples, y las no lineales anidadas.

En particular, las no lineales múltiples pueden ser convertidas en no lineales anidadas a través de hacer inmersión de dominio, agregando un parámetro de acumulación que guarda los resultados parciales de las llamadas recursivas.

En el caso de las no lineales anidadas la técnica involucra utilizar una pila para guardar los resultados parciales, de esta forma convirtiéndola en una función recursiva final, que por otro lado ya sabemos como convertir en iterativa.

### 3. Inducción sobre TADs

Te pedía básicamente plantear el esquema de inducción estructural para un TAD genérico, y justificar que se puede usar inducción en los TADs. Se puede hacer inducción sobre los reales? Y sobre el TAD racionales? Justifique.

### 3.1. Esquema de inducción

Dada una proposición  $P$  sobre un TAD  $T$ , si tomamos a  $g_1, \dots, g_k$  como los generadores del TAD que no toman ninguna instancia de argumento, a  $g_{k+1}, \dots, g_n$  como los generadores que sí toman una instancia previa, entonces tenemos el esquema de inducción sobre  $T$

$$\left( \bigwedge_{i=1}^k P(g_i) \wedge \bigwedge_{i=k+1}^n (\forall t : T)(P(t) \implies P(g_i(t))) \right) \implies (\forall t : T)P(t)$$

Observemos que esta fórmula no contempla generadores que reciban parámetros de otro tipo. En estos casos, lo único que hace falta es agregar una cuantificación universal por cada parámetro de otro tipo en los lugares adecuados. Suponiendo que  $T_1, \dots, T_n$  es el tipo de cada argumento que recibe cada generador, el esquema de inducción sería:

$$\left( \bigwedge_{i=1}^k [(\forall a : T_i)P(g_i(a))] \wedge \bigwedge_{i=k+1}^n (\forall t : T)(P(t) \implies ((\forall a : T_i)P(g_i(t, a)))) \right) \implies (\forall t : T)P(t)$$

### 3.2. Fundamentos teóricos

**Definición 1.** Sea  $S$  un conjunto,  $\star$  una relación binaria sobre  $S$ . Decimos entonces que  $\star$  es:

- *Reflexiva*  $\iff (\forall a \in S)(a \star a)$
- *Simétrica*  $\iff (\forall a, b \in S)(a \star b \implies b \star a)$
- *Transitiva*  $\iff (\forall a, b, c \in S)(a \star b \wedge b \star c \implies a \star c)$
- *Antisimétrica*  $\iff (\forall a, b \in S)(a \star b \wedge b \star a \implies a = b)$
- *Total*  $\iff (\forall a, b \in S)(a \star b \vee b \star a)$

**Definición 2.** Denominamos un orden parcial (poset) a un conjunto  $S$  junto con una relación binaria  $\prec$  definida sobre  $S$  tal que  $\prec$  es reflexiva, antisimétrica, y transitiva

**Definición 3.** En el caso en que  $\prec$  es transitiva, antisimétrica y total, decimos que  $\prec$  es un orden total.

**Definición 4.** Si tenemos  $\prec$  un orden total sobre  $S$  y para todo subconjunto no vacío de  $S$  hay un elemento mínimo, decimos que  $\prec$  es un buen orden sobre  $S$ .

**Teorema 1.** Principio de inducción bien fundada: si  $\prec$  define un buen orden sobre  $S$ ,  $P$  es un predicado sobre  $S$ , y

- $P$  vale para todos los elementos mínimos de  $S$  de acuerdo a  $\prec$ .
- $(\forall a \in S)[(\forall b \in S)(b \prec a \implies P(b))]$

Entonces,  $(\forall a \in S)P(a)$

*Principio de inducción bien fundada.* Por el contrarrecíproco, supongamos que  $\exists a \in S : \neg P(a)$ , para demostrar la validez del teorema, lo que necesitamos es mostrar que  $P$  no vale para todos los elementos mínimos de  $S$ , o que existe un elemento  $a$  de  $S$  tal que existe un  $b$  en  $S$  y  $\neg(b \prec a \implies P(a))$

Sabemos que el conjunto  $\{x : x \in S \wedge \neg P(x)\}$  es no vacío, y tiene un elemento mínimo  $m$  determinado por  $\prec$  porque  $\prec$  define un buen orden sobre  $S$ . Si  $m$  es un elemento mínimo de  $S$ , entonces no hay nada más que hacer, porque tendríamos que la primera condición es falsa. En el caso contrario, existen elementos de  $S$  que son menores a  $m$ , pero observemos que todos esos elementos menores a  $m$  sí cumplen  $P$  (porque sino serían  $m$ ), pero la segunda condición dice que entonces debería valer  $P(m)$ . Contradicción.  $\square$

Solo falta ver por qué existe un orden bien fundado sobre TADs. Notemos, primero, que las instancias de un TAD son numerables: todas las instancias de un TAD son generadas a través de un cantidad finita de aplicaciones de los generadores sobre una instancia inicial. Por lo tanto, existe una biyección con los naturales.