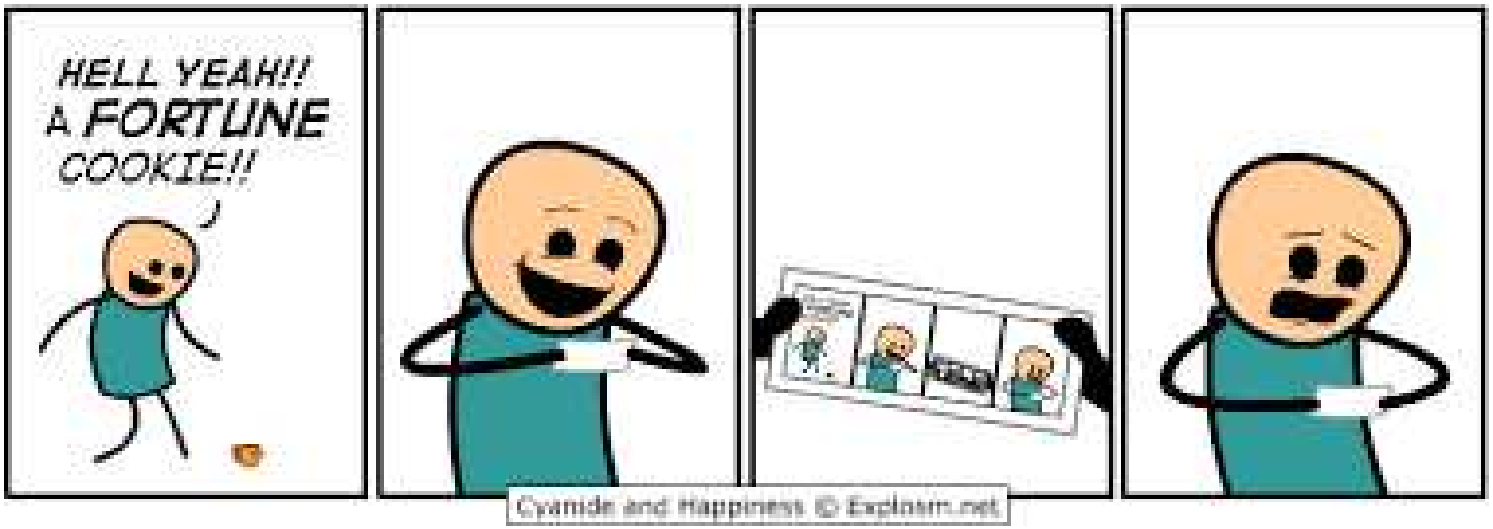
Diseño Recursivo



Yolanda Ortega Mallén

Objetivos

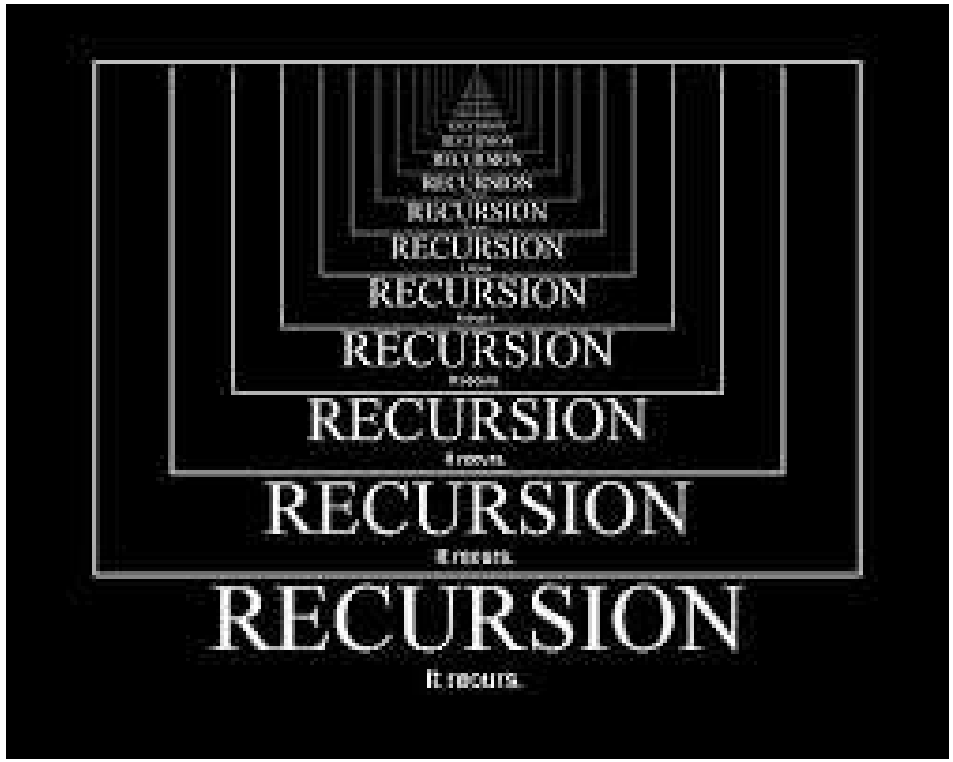
1. Realizar un diseño razonado de algoritmos recursivos.
2. Saber cómo añadir parámetros para mejorar el coste de los algoritmos recursivos.
3. Aprender a verificar la corrección de un algoritmo recursivo respecto a su especificación.
4. Aprender a obtener un algoritmo iterativo equivalente a uno recursivo.

Sumario

- Diseño de algoritmos recursivos.
- Verificación de algoritmos recursivos.
- Técnicas de inmersión.
- Transformación de algoritmos recursivos en iterativos.

Características de la recursión

¿Qué es? Permite que un procedimiento o función haga referencia a sí mismo dentro de su definición.



¿Cómo funciona? Resuelve un problema P sobre unos datos D suponiendo que

Recursión vs iteración

Ejemplo: Potencia

Problema: $P \equiv m^n$ Datos: $D \equiv (m,n)$

Método iterativo

$m^n = m * m * \ldots^{(n} * m.$
Nuevo problema:
$P' \equiv$ multiplicar dos enteros.

fun potencia$(m,n : ent)$ **dev** $p : ent$
var $x : ent$
$\langle x,p \rangle := \langle 0,1 \rangle$;
mientras $(x \neq n)$ **hacer**
$p := p * m$;
$x := x + 1$
fmientras
ffun

Método recursivo

$m^n = m^{n-1} * m.$
Mismo problema para datos más pequeños:
$D' \equiv (m,n-1).$

fun potencia$(m,n : ent)$ **dev** $p : ent$
casos
$n = 0 \rightarrow p := 1$
$\square$ $n > 0 \rightarrow p :=$ $m *$ potencia$(m,n-1)$
fcasos
ffun

Esquema general

$\{ P(\overline{x}) \}$
fun `f-rec`$(\overline{x} : T_1)$ **dev** $\langle \overline{y} : T_2 \rangle$
casos
$B_t(\overline{x}) \rightarrow \overline{y} := \texttt{triv}(\overline{x})$
$\square$ $B_{nt}(\overline{x}) \rightarrow \overline{y} := \texttt{comp}(\texttt{f-rec}(\texttt{s}(\overline{x})), \overline{x})$
fcasos
ffun
$\{ Q(\overline{x}, \overline{y}) \}$

Recursión lineal única llamada recursiva.

Recursión final `comp` no realiza acciones (devuelve el primer argumento).

Recursón múltiple más de una llamada recursiva.

Diseño y verificación de algoritmos recursivos

Los pasos a seguir son:

1. Especificación formal del algoritmo.
2. Análisis por casos, descomposición.
3. Composición de resultados.
4. Verificación formal de la corrección.
5. Estudio del coste.

Análisis por casos y composición

¿Cómo descomponer los datos para calcular la solución a partir de la solución al mismo problema para datos más pequeños?

Análisis de casos: triviales o básicos

Reducir el tamaño de los problemas en los casos recursivos y acercarse al caso trivial.

Clasificación exhaustiva cubrir todos los casos permitidos por la precondición.

Casos excluyentes no ambigüedad.

Potencia

$$\begin{array}{c|c} n = 0 & m^0 = 1 \\ \hline n > 0 & m^n = m \cdot m^{n-1} \end{array}$$

Corrección

Demostrar que para todo posible dato de entrada $\overline{x}$ que cumpla $P(\overline{x})$, se cumple $Q(\overline{x}, \texttt{f-rec}(\overline{x}))$.

1. Se cubren todos los casos:

$$P(\overline{x}) \Rightarrow B_t(\overline{x}) \ \vee \ B_{nt}(\overline{x})$$

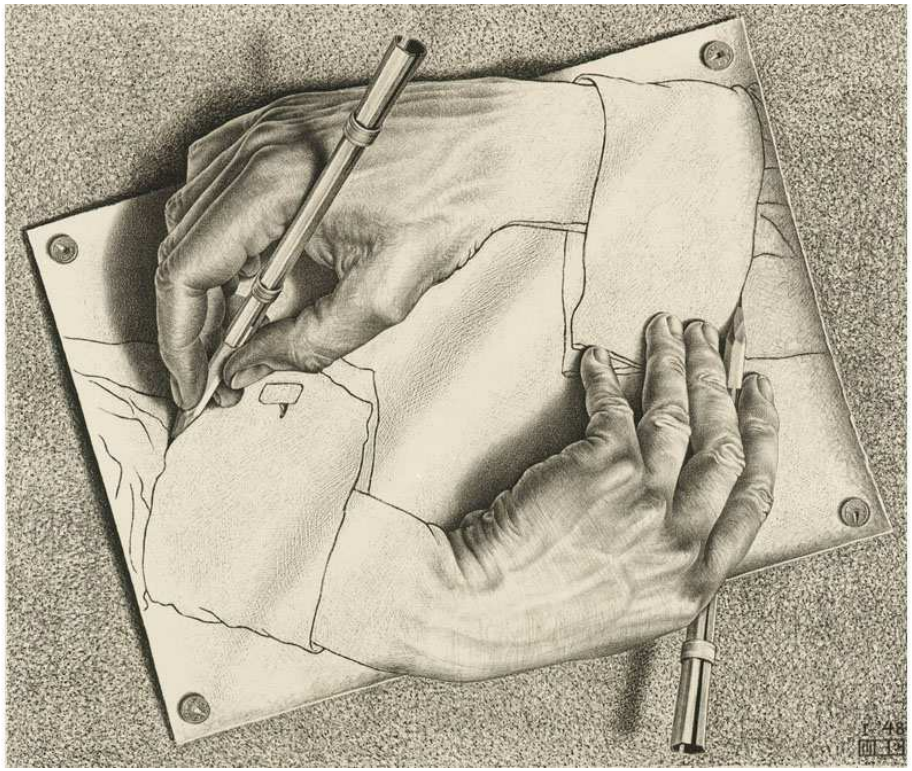
2. El caso trivial es correcto:

$$P(\overline{x}) \ \wedge \ B_t(\overline{x}) \Rightarrow Q(\overline{x}, \texttt{triv}(\overline{x}))$$

3. La función recursiva es invocada siempre en estados que satisfacen su precondición:

$$P(\overline{x}) \ \wedge \ B_{nt}(\overline{x}) \Rightarrow P(\texttt{s}(\overline{x}))$$

Terminación



5. Existe una función de tamaño t tal que

$$P(\overline{x}) \Rightarrow t(\overline{x}) \geq 0$$

$$\dots$$

$$P(x) \wedge B_{nt}(x) \Rightarrow t(\mathbf{s}(x)) < t(x)$$

Ejemplo: potencia

$\{ n \geq 0 \}$
fun potencia$(m, n : ent)$ **dev** $p : ent$
  casos
    $n = 0 \rightarrow p := 1$
  $\square$ $n > 0 \rightarrow$
    $p := m * \texttt{potencia}(m, n-1)$
  fcasos
ffun
$\{ p = m^n \}$

$\overline{x} = \langle m, n \rangle$
$\overline{y} = p$
$P(m,n) \equiv n \geq 0$
$Q(m,n,p) \equiv p = m^n$

$B_t(m,n) \equiv n = 0$
$B_{nt}(m,n) \equiv n > 0$
$\mathtt{s}(m,n) = \langle m, n-1 \rangle$
$\mathtt{triv}(m,n) = 1$
$\mathtt{comp}(p', m, n) = m * p'$
$t(m,n) = n$

1. $n \geq 0 \Rightarrow n = 0 \ \vee \ n > 0$
2. $n \geq 0 \wedge n = 0 \Rightarrow 1 = m^n$
3. $n \geq 0 \wedge n > 0 \Rightarrow n - 1 \geq 0$
4. $n \geq 0 \wedge n > 0 \wedge p' = m^{n-1} \Rightarrow m * p' = m^n$

Ejemplo: máximo común divisor

$\{ a > 0 \ \wedge \ b > 0 \}$
fun euclides$(a, b : nat)$ **dev** $mcd : nat$
  casos
    $a = b \ \rightarrow \ mcd := a$
  $[]$ $a > b \ \rightarrow \ mcd :=$ euclides$(a - b, b)$
  $[]$ $a < b \ \rightarrow \ mcd :=$ euclides$(a, b - a)$
  fcasos
ffun
$\{ mcd = \text{mcd}(a, b) \}$

Propiedades del máximo común divisor:

- $mcd(a,b) = mcd(a-b,b)$ si $a > b$.
- $mcd(a,b) = mcd(a,b-a)$ si $a < b$.
- $mcd(a,b) = a$ si $a = b$

1. $a > 0 \ \wedge \ b > 0 \Rightarrow a = b \ \vee \ a > b \ \vee \ a < b$
2. $a > 0 \wedge b > 0 \wedge a = b \Rightarrow a = mcd(a,b)$
3. $a > 0 \wedge b > 0 \wedge a > b \Rightarrow a - b > 0 \wedge b > 0$
   $a > 0 \wedge b > 0 \wedge a < b \Rightarrow a > 0 \wedge b - a > 0$
4. $a > 0 \wedge b > 0 \wedge a > b \wedge mcd' = mcd(a-b,b) \Rightarrow mcd' = mcd(a,b)$
   $a > 0 \wedge b > 0 \wedge a < b \wedge mcd' = mcd(a,b-a) \Rightarrow mcd' = mcd(a,b)$

$a > 0 \wedge b > 0 \wedge a < b \Rightarrow a + b - a < a + b$

Análisis del coste

Recurrencias: funciones de coste recursivas.

fun `factorial`$(n : nat)$ **dev** $f : nat$
  casos
    $n = 0 \rightarrow f := 1$
  $\square$ $n > 0 \rightarrow$
    $f := n * \texttt{factorial}(n-1)$
  fcasos
ffun

Recurrencia

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

Para obtener el orden de complejidad de $T(n)$:

Despliegue obtener fórmula explícita de $T(n)$.

Teoremas disminución del tamaño del problema por sustracción o por división.

Despliegue de recurrencias

El objetivo es conseguir una fórmula explícita (en función de n) de $T(n)$.

Despliegue Sustituir T por la parte derecha de la ecuación.
Repetir hasta encontrar una fórmula que dependa del número de despliegues (o llamadas recursivas) i.

Postulado Obtener el valor de i que hace desaparecer T (caso básico).
En la fórmula, sustituir i por ese valor para obtener la fórmula explícita T^*, que solo depende de n.

Comprobación Demostrar por inducción que $T = T^*$.

Factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

$$\begin{array}{lcll} T(n) & \overset{1}{=} & T(n-1) + k_2 & \\ & \overset{2}{=} & T(n-2) + k_2 + k_2 & = \quad T(n-2) + 2 \cdot k_2 \\ & \overset{3}{=} & T(n-3) + k_2 + 2 \cdot k_2 & = \quad T(n-3) + 3 \cdot k_2 \\ & \vdots & & \\ & \overset{i}{=} & T(n-i) + i \cdot k_2 & \\ & \vdots & & \\ & \overset{n}{=} & T(0) + n \cdot k_2 & = \quad n \cdot k_2 + k_1 = T^*(n) \in \Theta(n) \end{array}$$

$\forall n \geq 0 \,.\, T(n) = T^*(n)$

Caso base $T(0) = k_1 = T^*(0)$

- Caso directo: coste constante
- Preparación de llamadas y combinación de resultados: coste polinómico

Teorema de la resta: Descomposición restando una cantidad constante

$$T(n) = \begin{cases} k_0 & \text{si } 0 \leq n < b \\ a \cdot T(n-b) + k_1 \cdot n^k & \text{si } n \geq b \end{cases}$$

$$T(n) \in \begin{cases} \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

Teorema de la división: Descomposición dividiendo por una cantidad constante $b \geq 2$

$$T(n) = \begin{cases} k_0 & \text{si } 0 \leq n < b \\ a \cdot T(\frac{n}{b}) + k_1 \cdot n^k & \text{si } n \geq b \end{cases}$$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

2. El coste es menor cuanto más pequeñas son a y k y más grande es b.

División entera

$\{ a \geq 0 \ \wedge \ b > 0 \}$
fun `dividir`$(a, b : ent)$ **dev** $\langle q, r : ent \rangle$
$\{ a = b * q + r \ \wedge \ 0 \leq r < b \}$

Análisis por casos y composición

- $a < b \longrightarrow q = 0$ y $r = a$ caso básico
- $a = b \longrightarrow q = 1$ y $r = 0$ caso básico
- $a > b \ \wedge \ a = b * q + r \Rightarrow a - b = b * (q - 1) + r$ caso recursivo

$a < b$	$q = 0$ y $r = a$
$a \geq b$	$q = q' + 1$ y $r = r'$ donde $\langle q', r' \rangle =$ `dividir`$(a - b, b)$

Ejemplo: División entera

fun `dividir`$(a,b : ent)$ **dev** $\langle q,r : ent \rangle$
  casos
    $a < b \rightarrow \langle q,r \rangle := \langle 0,a \rangle$
  $\square$ $a \geq b \rightarrow \langle q,r \rangle := \texttt{dividir}(a-b,b)$;
    $q := q+1$
  fcasos
ffun

Coste depende de a y de b; tamaño: $n = a \text{ div } b$
$(a-b) \text{ div } b = (a \text{ div } b) - 1$

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

Ejemplo: División entera

Mayor eficiencia si dividimos el tamaño del problema.

- dividiendo a
- multiplicando b

$$\langle q', r' \rangle = \texttt{dividir}(a, 2b) \Rightarrow a = (2b) * q' + r' \ \wedge \ 0 \leq r' < 2b$$

Buscamos $\langle q, r \rangle$ tal que $a = b * q + r \ \wedge \ 0 \leq r < b$:

- $r' < b \longrightarrow \langle q, r \rangle = \langle 2q', r' \rangle$
- $b \leq r' < 2b \Rightarrow 0 \leq r' - b < b$

$$a = (2b) * q' + b + r' - b = b * (2q' + 1) + (r' - b)$$

$a < b$	$q = 0$ y $r = a$
$a \geq b$	$\texttt{dividir}(a - 2b)$

Ejemplo: División entera

fun `dividir-efic`$(a,b : ent)$ **dev** $\langle q,r : ent \rangle$

casos

$$\begin{array}{rcl} a < b & \rightarrow & \langle q,r \rangle := \langle 0,a \rangle \\ \square\ a \geq b & \rightarrow & \langle q,r \rangle := \texttt{dividir-efic}(a,2*b)\ ; \end{array}$$

casos

$$\begin{array}{rcl} r < b & \rightarrow & q := 2*q \\ \square\ r \geq b & \rightarrow & \langle q,r \rangle := \langle 2*q+1, r-b \rangle \end{array}$$

fcasos

fcasos

ffun

Tamaño: $n = a \text{ div } b$ $\quad a \text{ div } (2b) = (a \text{ div } b) \text{ div } 2$

$$T(n) = \begin{cases} k_1 & n = 0 \\ T(n \text{ div } 2) + k_2 & n > 0 \end{cases}$$

Verificar la corrección de los dos algoritmos para la división entera.

Búsqueda binaria

$\{\ ord(V,1,N)\ \wedge\ N \geq 0\ \}$
fun búsqueda-binaria-rec$(V[1..N]$ **de** $ent, A : ent)$ **dev** $\langle b : bool, p : nat \rangle$
$\{(b \rightarrow 1 \leq p \leq N\ \wedge\ V[p] = A)\ \wedge$
$(\neg b \rightarrow 1 \leq p \leq N+1\ \wedge\ V[1..p) \prec A\ \wedge\ A \prec V[p..N])\ \}$

$$ord(V,a,b) \equiv (\forall k : a \leq k < b : V[k] < V[k+1])$$
$$V[a..b) \prec x \equiv (\forall i : a \leq i < b : V[i] < x)$$
$$x \prec V[a..b] \equiv (\forall i : a \leq i \leq b : x < V[i])$$

Generalización: búsqueda en cualquier segmento $V[c..f]$.

$\{\ P \equiv ord(V,c,f)\ \wedge\ 1 \leq c \leq f+1 \leq N+1\ \}$
fun búsqueda-binaria-rec$(V[1..N]$ **de** $ent, A : ent, c, f : nat)$ **dev** $\langle b : bool, p : nat \rangle$
$\{\ Q \equiv (b \rightarrow c \leq p \leq f\ \wedge\ V[p] = A)\ \wedge$

Ejemplo: Búsqueda binaria

fun búsqueda-binaria-rec$(V[1..N]$ **de** $ent, A : ent, c, f : nat)$ **dev** $\langle b : bool, p : nat \rangle$
  si $c > f$ **entonces** $\langle b, p \rangle := \langle$ falso$, c \rangle$
  si no
    $m := (c + f)$ div 2 ;
    casos
      $A < V[m] \rightarrow \langle b, p \rangle :=$ búsqueda-binaria-rec$(V, A, c, m-1)$
    [] $A = V[m] \rightarrow \langle b, p \rangle := \langle$ cierto$, m \rangle$
    [] $A > V[m] \rightarrow \langle b, p \rangle :=$ búsqueda-binaria-rec$(V, A, m+1, f)$
    fcasos
  fsi
ffun

Correcto para cualquier m tal que $c \leq m \leq f$.
$m := (c + f)$ div 2 es la más eficiente.

Ejemplo: Búsqueda binaria

1. Se cubren todos los casos:
   - $P \Rightarrow c > f \ \vee \ c \leq f$
   - $c \leq f \wedge m = (c+f) \text{ div } 2 \Rightarrow c \leq m \leq f$

     $P \wedge (c \leq m \leq f) \Rightarrow A < V[m] \ \vee \ A = V[m] \ \vee \ A > V[m]$
2. Los casos triviales son correctos:
   - $P \wedge (c > f) \Rightarrow (\texttt{falso} \rightarrow c \leq c \leq f \ \wedge \ V[c] = A) \ \wedge$

     $(\neg\texttt{falso} \rightarrow c \leq c \leq f+1 \ \wedge \ V[c..c) \prec A \ \wedge \ A \prec V[c..f])$
   - $P \ \wedge \ (c \leq m \leq f) \wedge A = V[m] \Rightarrow$

     $(\texttt{cierto} \rightarrow c \leq m \leq f \ \wedge \ V[m] = A) \ \wedge$

     $(\neg\texttt{cierto} \rightarrow c \leq m \leq f+1 \ \wedge \ V[c..m) \prec A \ \wedge \ A \prec V[m..f])$
3. La función recursiva es invocada en estados que satisfacen su precondición:
   - $ord(V,c,f) \wedge (1 \leq c \leq f+1 \leq N+1) \wedge (c \leq m \leq f) \Rightarrow$

     $ord(V,c,m-1) \wedge (1 \leq c \leq m \leq N+1)$

Ejemplo: Búsqueda binaria

4. Los pasos de inducción son correctos:
   - $ord(V,c,f) \wedge (1 \leq c \leq f+1 \leq N+1) \wedge (c \leq m \leq f) \wedge (A < V[m]) \wedge$
     $(b' \rightarrow c \leq p' \leq m-1 \wedge V[p'] = A) \wedge$
     $(\neg b' \rightarrow c \leq p' \leq m \wedge V[c..p') \prec A \wedge A \prec V[p'..m-1])$
     $\Rightarrow (b' \rightarrow c \leq p' \leq f \wedge V[p'] = A) \wedge$
     $(\neg b' \rightarrow c \leq p' \leq f+1 \wedge V[c..p') \prec A \wedge A \prec V[p'..f])$
   - $ord(V,c,f) \wedge (1 \leq c \leq f+1 \leq N+1) \wedge (c \leq m \leq f) \wedge (A > V[m]) \wedge$
     $(b' \rightarrow m+1 \leq p' \leq f \wedge V[p'] = A) \wedge$
     $(\neg b' \rightarrow m+1 \leq p' \leq f+1 \wedge V[m+1..p') \prec A \wedge A \prec V[p'..f])$
     $\Rightarrow (b' \rightarrow c \leq p' \leq f \wedge V[p'] = A) \wedge$
     $(\neg b' \rightarrow c \leq p' \leq f+1 \wedge V[c..p') \prec A \wedge A \prec V[p'..f])$
5. Tamaño: $t = f - c + 1$

$$ord(V,c,f) \wedge (1 \leq c \leq f+1 \leq N+1) \Rightarrow (f-c+1 \geq 0)$$

6. El valor de t decrece al hacer las llamadas recursivas:

$$\Rightarrow (t - (m+1) + 1 < f - c + 1)$$

Técnicas de inmersión

Cuando la función especificada `f` no admite una descomposición recursiva, lo intentamos con una función más general: con más parámetros (de entrada) o más resultados, tal que para ciertos valores calcule lo mismo que `f`.

$\{ N \geq 0 \}$
fun `suma-vector`$(V[0..N)$ **de** *ent*$)$ **dev** s : *ent*
$\{ s = (\sum i : 0 \leq i < N : V[i]) \}$

Solución: debilitar la postcondición, sustituyendo la constante N por una variable n, que se añade como parámetro.

$\{ N \geq 0 \ \wedge \ 0 \leq n \leq N \}$
fun `gsuma-vector`$(V[0..N)$ **de** *ent*, n : *nat*$)$ **dev** s : *ent*
$\{ s = (\sum i : 0 \leq i < n : V[i]) \}$

$$\texttt{gsuma-vector}(V, N) = \texttt{suma-vector}(V)$$

$$\overline{\texttt{gsuma-vector}}(V, N, \ldots) = \overline{\texttt{suma-vector}}(V) \ \ldots$$

$\{ N \geq 0 \ \wedge \ 0 \leq n \leq N \}$

fun `gsuma-vector`$(V[0..N)$ **de** *ent*$, n$: *nat*$)$ **dev** s : *ent* $\quad \{\Theta(n)\}$

casos

$$\begin{array}{rcl} n = 0 & \rightarrow & s := 0 \\ \square \ n > 0 & \rightarrow & s := \texttt{gsuma-vector}(V, n-1) \ ; \\ & & s := s + V[n-1] \end{array}$$

fcasos

ffun

$\{ s = (\sum i : 0 \leq i < n : V[i]) \}$

Inmersión de f en g

$\{ P(\overline{x}) \}$
fun f$(\overline{x})$ **dev** $\overline{y}$
$\{ Q(\overline{x}, \overline{y}) \}$

y especificamos e implementamos una función g más general.

$\{ P'(\overline{x}, \overline{w}) \}$
fun g$(\overline{x}, \overline{w})$ **dev** $\langle \overline{y}, \overline{z} \rangle$
$\{ Q'(\overline{x}, \overline{w}, \overline{y}, \overline{z}) \}$

tal que, bajo ciertas condiciones $R(\overline{x}, \overline{w})$, se cumple que g se comporta como f

$$P'(\overline{x}, \overline{w}) \wedge \underbrace{R(\overline{x}, \overline{w})}_{\overline{w}=\overline{c}} \wedge Q'(\overline{x}, \overline{w}, \overline{y}, \overline{z}) \Rightarrow Q(\overline{x}, \overline{y})$$

f es la función sumergida y g la función inmersora.

Ejemplo: Raíz cuadrada entera

$\{ P \equiv n \geq 0 \}$
fun `raíz-ent`$(n : ent)$ **dev** $r : ent$
$\{ Q \equiv r^2 \leq n < (r+1)^2 \}$

Inmersión: sustituir en Q la constante 1 por la variable a.

$Q' \equiv r^2 \leq n < (r+a)^2$

$Q' \wedge a = 1 \Rightarrow Q$

$\{ P' \equiv n \geq 0 \wedge a \geq 1 \}$
fun `graíz-ent`$(n, a : ent)$ **dev** $r : ent$
$\{ Q' \equiv r^2 \leq n < (r+a)^2 \}$

Ejemplo: Raíz cuadrada entera

Cuando a es *suficientemente* grande ($n < a^2$), la solución (trivial) será $r = 0$.

Podemos intentar aumentar a, pasando a $2a$.

¿Existe relación entre `graíz-ent`(n, a) y `graíz-ent`$(n, 2a)$?

La llamada recursiva devolverá r' tal que

$$r'^2 \leq n \ \wedge \ n < (r' + 2a)^2$$

Si r' cumpliera además $n < (r' + a)^2$, entonces valdría como resultado.

En caso contrario tenemos

$$(r' + a)^2 \leq n \wedge n < (r' + a + a)^2$$

Ejemplo: Raíz cuadrada entera

$\{ P' \equiv n \geq 0 \ \wedge \ a \geq 1 \}$

fun `graíz-ent`$(n, a : ent)$ **dev** $r : ent$

casos

$$\begin{array}{rcl} a^2 > n & \rightarrow & r := 0 \\ \square\ a^2 \leq n & \rightarrow & r := \texttt{graíz-ent}(n, 2 * a)\ ; \\ & & \textbf{casos} \\ & & \quad n < (r+a)^2 \ \rightarrow \ \texttt{nada} \\ & & \square\ n \geq (r+a)^2 \ \rightarrow \ r := r + a \\ & & \textbf{fcasos} \end{array}$$

fcasos

ffun

$\{ Q' \equiv r^2 \leq n < (r+a)^2 \}$

Tamaño: $m = n/a^2$

$$n/(2a)^2 = (n/a^2)/4$$

`graíz-ent` tiene coste en $\Theta(\log m)$ y `raíz-ent` tiene coste en $\Theta(\log n)$.

Inmersión final

Cuando la función inmersora es recursiva final sus resultados valen como resultados de la función sumergida:

$$Q' \Rightarrow Q$$

y no depende de los parámetros adicionales $\overline{w}$.

¿Para qué sirven en este caso los parámetros adicionales?

Para acumular parte de los resultados y no tener que realizar el camino ascendente.

Ejemplo: Suma de los elementos de un vector

$\{ P' \equiv 0 \leq n \leq N \ \wedge \ w = (\sum i : 0 \leq i < n : V[i]) \}$
fun `gfsuma-vector`$(V[0..N)$ **de** *ent*$, n$: *nat*$, w$: *ent*$)$ **dev** s : *ent*
$\{ Q' \equiv s = (\sum i : 0 \leq i < N : V[i]) \}$

Como $P' \ \wedge \ n = N \Rightarrow Q'^{w}_{s}$ consideramos el caso trivial $B_t \equiv n = N$, en el cual se devuelve como resultado el acumulador.

Llamada inicial: $n = 0$ y $w = 0$.

Sucesor: $\mathtt{s}(n) = n + 1$.

La precondición se debe cumplir al hacer la llamada recursiva:

$$w = (\sum i : 0 \leq i < n+1 : V[i]) = (\sum i : 0 \leq i < n : V[i]) + V[n]$$

fun `gfsuma-vector`$(V[0..N)$ **de** *ent*$, n$: *nat*$, w$: *ent*$)$ **dev** s : *ent*
casos

Inmersión por razones de eficiencia

Añadir más parámetros (de entrada o salida) para que la función sea más eficiente.

Calcular expresiones costosas mediante parámetros acumuladores.

Inmersión de parámetros se necesita una expresión compleja e antes de la llamada recursiva.
Nuevo parámetro p de entrada, y se añade $p = e$ a la precondición.

Inmersión de resultados se necesita una expresión compleja e después de la llamada recursiva.
Nuevo parámetro de salida p, y se añade $p = e$ a la postcondición.

Ejemplo: Raíz cuadrada entera

Inmersión de parámetros para calcular $(r+a)^2 = r^2 + a^2 + 2ra$.

$\{ n \geq 0 \wedge a \geq 1 \wedge ac = a^2 \}$
fun ggraíz-ent$(n, a, ac : ent)$ **dev** $r : ent$
  casos
    $ac > n \rightarrow r := 0$
    $\square$ $ac \leq n \rightarrow r :=$ ggraíz-ent$(n, 2*a, 4*ac)$;
      casos
        $n < r^2 + ac + 2*r*a \rightarrow$ nada
        $\square$ $n \geq r^2 + ac + 2*r*a \rightarrow r := r + a$
      fcasos
  fcasos
ffun
$\{ r^2 \leq n < (r+a)^2 \}$

Ejemplo: Raíz cuadrada entera

Inmersión de resultados para calcular r^2.

$\{ n \geq 0 \wedge a \geq 1 \wedge ac = a^2 \}$
fun `gggraíz-ent`$(n, a, ac : ent)$ **dev** $\langle r, rc : ent \rangle$
casos
$ac > n \rightarrow \langle r, rc \rangle := \langle 0, 0 \rangle$
$\square$ $ac \leq n \rightarrow \langle r, rc \rangle :=$ `gggraíz-ent`$(n, 2 * a, 4 * ac)$;
$e := rc + ac + 2 * r * a$; $\{ e = (r+a)^2 \}$
casos
$n < e \rightarrow$ `nada`
$\square$ $n \geq e \rightarrow \langle r, rc \rangle := \langle r + a, e \rangle$
fcasos
fcasos
ffun
$\{ r^2 \leq n < (r+a)^2 \wedge rc = r^2 \}$

Función de Fibonacci eficiente

$\{$ `cierto` $\}$
fun `gfibonacci`$(n : nat)$ **dev** $\langle f, g : nat \rangle$
casos

$$\begin{array}{rcl} n = 0 & \rightarrow & \langle f, g \rangle := \langle 0, 1 \rangle \\ \square\ n > 0 & \rightarrow & \langle f, g \rangle := \texttt{gfibonacci}(n-1)\ ; \\ & & \langle f, g \rangle := \langle g, f+g \rangle \end{array}$$

fcasos
ffun
$\{ f = fib_n \wedge g = fib_{n+1} \}$

$T(n) \in \Theta(n)$

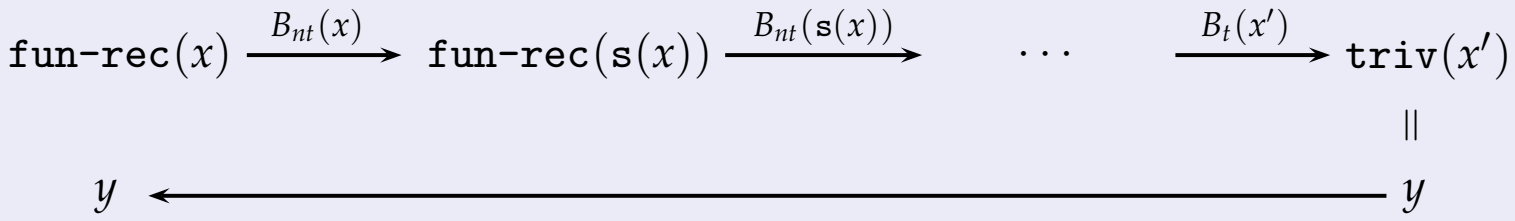
Transformación de recursivo a iterativo

- El lenguaje disponible no soporta recursión.
- Reducción del coste adicional en *tiempo*
  ↪ llamada a subprograma y paso de parámetros.
- Reducción del coste adicional en *memoria*
  ↪ pila de activaciones.

Pero la versión iterativa es menos legible y modificable.

Transformación automatizada (compilador) en algunos casos
↪ versión iterativa optimizada en memoria

Transformación de recursivo final a iterativo



$\{ P(\overline{x}) \}$
fun `f-rec`$(\overline{x})$ **dev** $\overline{y}$
  casos
    $B_t(\overline{x}) \rightarrow \overline{y} := \texttt{triv}(\overline{x})$
    $\square\ B_{nt}(\overline{x}) \rightarrow \overline{y} := \texttt{f-rec}(\texttt{s}(\overline{x}))$
  fcasos
ffun
$\{ Q(\overline{x},\overline{y}) \}$

$\{ P(\overline{x}) \}$
fun `f-it`$(\overline{x})$ **dev** $\overline{y}$
var $\overline{x}'$
  $\overline{x}' := \overline{x}$;
  $\{ INV(\overline{x}',\overline{x}) \}$
  mientras $B_{nt}(\overline{x}')$ **hacer**
    $\overline{x}' := \texttt{s}(\overline{x}')$
  fmientras ;
  $\overline{y} := \texttt{triv}(\overline{x}')$
ffun
$\{ Q(\overline{x},\overline{y}) \}$

Verificar la versión iterativa

Ejemplo: Suma de los elementos de un vector

fun gfsuma-vector$(V[0..N)$ **de** *ent*, n : *nat*, w : *ent*$)$ **dev** s : *ent*

casos

$$\begin{array}{rcl} n = N & \rightarrow & s := w \\ \square\ n < N & \rightarrow & s := \texttt{gfsuma-vector}(V, n+1, w + V[n]) \end{array}$$

fcasos

ffun

fun gfsuma-vector-it$(V[0..N)$ **de** *ent*, n : *nat*, w : *ent*$)$ **dev** s : *ent*

var n' : *nat*, w' : *ent*

$\langle n', w' \rangle := \langle n, w \rangle$;

mientras $n' < N$ **hacer**

$\langle n', w' \rangle := \langle n'+1, w' + V[n'] \rangle$

fmientras ;

$s := w'$

ffun

fun suma-vector-it$(V[0..N)$ **de** *ent*$)$ **dev** s : *ent*

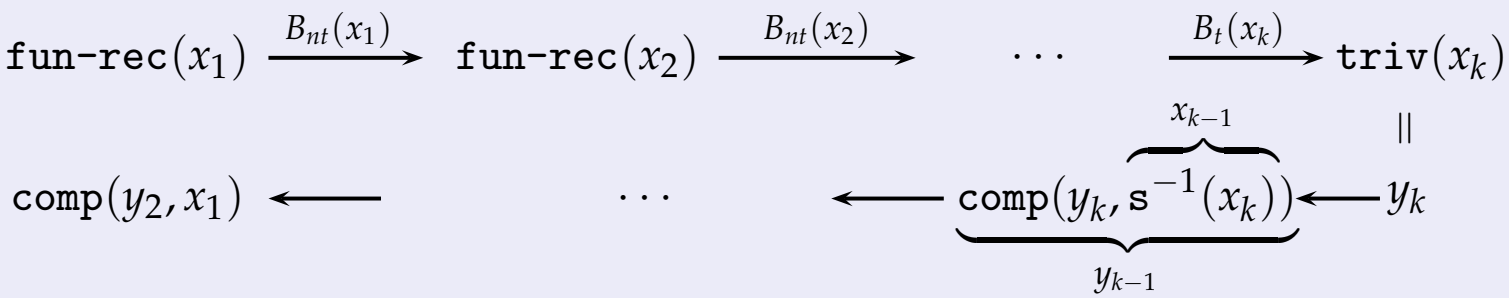
var n : *nat*

Ejemplo: Máximo común divisor

fun mcd-rec$(a, b : ent)$ **dev** $mcd : ent$
  casos
    $a = b \rightarrow mcd := a$
  [] $a > b \rightarrow mcd :=$ mcd-rec$(a - b, b)$
  [] $a < b \rightarrow mcd :=$ mcd-rec$(a, b - a)$
  fcasos
ffun

fun mcd-it$(a, b : ent)$ **dev** $mcd : ent$
var $a', b' : ent$
  $\langle a', b' \rangle := \langle a, b \rangle$;
  mientras $a' \neq b'$ **hacer**
    casos
      $a' > b' \rightarrow a' := a' - b'$
    [] $a' < b' \rightarrow b' := b' - a'$

Transformación de recursivo (lineal) no final a iterativo



$\{ P(\overline{x}) \}$
fun f-rec$(\overline{x})$ **dev** $\overline{y}$
  casos
    $B_t(\overline{x}) \rightarrow \overline{y} := \texttt{triv}(\overline{x})$
    $\square$ $B_{nt}(\overline{x}) \rightarrow$
      $\overline{y} := \texttt{comp}(\texttt{f-rec}(\texttt{s}(\overline{x})), \overline{x})$
  fcasos
ffun

fun f-it$(\overline{x})$ **dev** $\overline{y}$
var $\overline{x}'$
  $\overline{x}' := \overline{x}$;
  $\{ INV_1(\overline{x}', \overline{x}) \}$
  mientras $B_{nt}(\overline{x}')$ **hacer**
    $\overline{x}' := \texttt{s}(\overline{x}')$
  fmientras ;
  $\overline{y} := \texttt{triv}(\overline{x}')$;
  $\{ INV_2(\overline{x}', \overline{x}, \overline{y}) \}$
  mientras $\overline{x}' \neq \overline{x}$ **hacer**
    ...
ffun

$$INV_1(\overline{x}', \overline{x}) \equiv P(\overline{x}') \wedge suc(\overline{x}', \overline{x})$$

$$suc(\overline{x}', \overline{x}) \equiv (\exists i : 0 \leq i : \overline{x}' = \mathbf{s}^i(\overline{x}) \wedge (\forall j : 0 \leq j < i : B_{nt}(\mathbf{s}^j(\overline{x})) \wedge P(\mathbf{s}^j(\overline{x}))))$$

$$INV_2(\overline{x}', \overline{x}, \overline{y}) \equiv INV_1(\overline{x}', \overline{x}) \wedge Q(\overline{x}', \overline{y})$$

Ejercicio

Verificar la versión iterativa.

Ejemplo: Suma de los elementos de un vector

fun gsuma-vector$(V[0..N)$ **de** *ent*, n : *nat*$)$ **dev** s : *ent*

casos

$$\begin{array}{rcl} n = 0 & \rightarrow & s := 0 \\ \square\ n > 0 & \rightarrow & s := \texttt{gsuma-vector}(V, n-1)\ ; \\ & & s := s + V[n-1] \end{array}$$

fcasos

ffun

fun gsuma-vector-it$(V[0..N)$ **de** *ent*, n : *nat*$)$ **dev** s : *ent*

var n' : *nat*

$n' := n$;

mientras $n' > 0$ **hacer**

$n' := n' - 1$

fmientras ;

} equivalente a $n' := 0$

$s := 0$;

mientras $n' \neq n$ **hacer**

...

Ejemplo: División entera

fun `dividir-rec`$(a, b : ent)$ **dev** $\langle q, r : ent \rangle$

casos

$$\begin{array}{rcl} a < b & \rightarrow & \langle q, r \rangle := \langle 0, a \rangle \\ \square\ a \geq b & \rightarrow & \langle q, r \rangle := \texttt{dividir-rec}(a - b, b) \\ & & q := q + 1 \end{array}$$

fcasos

ffun

fun `dividir-it`$(a, b : ent)$ **dev** $\langle q, r : ent \rangle$

var $a', b' : ent$

$a' := a \; ; \;\; b' := b \; ;$

mientras $a' \geq b'$ **hacer**

$a' := a' - b'$

fmientras ;

$\langle q, r \rangle := \langle 0, a' \rangle \; ;$

¿Si no existe la función inversa del sucesor, s^{-1}?

s	s^{-1}
$+1$	-1
-1	$+1$
$*2$	div 2
div 2	no hay

Resolvemos este problema utilizando una pila:

fun f-it$(\overline{x})$ **dev** $\overline{y}$
var $\overline{x'}, p$: *pila*
  $\langle \overline{x'}, p \rangle := \langle \overline{x}, \texttt{pila-vacía}() \rangle$;
  mientras $B_{nt}(\overline{x'})$ **hacer**
    $\langle \overline{x'}, p \rangle := \langle \texttt{s}(\overline{x'}), \texttt{apilar}(p, \overline{x'}) \rangle$
  fmientras ;
  $\overline{y} := \texttt{triv}(\overline{x'})$;
  mientras ¬es-pila-vacía(p) **hacer**
    ...
ffun

Ejemplo: Sumar dígitos

fun suma-dígitos$(n : nat)$ **dev** $s : nat$
  casos
    $n < 10 \rightarrow s := n$
  $\square$ $n \geq 10 \rightarrow s :=$ suma-dígitos$(n \text{ div } 10) + (n \text{ mód } 10)$
  fcasos
ffun

fun suma-dígitos-it$(n : nat)$ **dev** $s : nat$
var $n' : nat, p : pila[nat]$
  $n' := n$;
  $p :=$ pila-vacía$()$;
  mientras $n' \geq 10$ **hacer**
    apilar(p, n') ;
    $n' := n' \text{ div } 10$
  fmientras ;
  $s := n'$