APUNTES

PROGRAMACIÓNII

Curso 1999 – 2000

© *José Manuel Carrillo*

NOTA: El curso pasado (1999-2000) me enfrenté a esta asignatura con los mismos prejuicios y reticencias que todos tenemos debido a los comentarios que hemos oído acerca de su dificultad. Quizá la tomé como un desafío y por eso le dediqué muchas más horas que a cualquier otra asignatura.

Como es lógico al principio no entendía nada. La primera luz apareció al consultar la página de Jerónimo Quesada (*http://www.bitabit.com)*. Es más que interesante que la visitéis sobre todo a la hora de acometer la práctica.

Cuando conseguí entender de qué iba la asignatura volví sobre mis pasos y empecé a estudiarla de nuevo.

Lo que tienes ahora en tus manos es el resultado de esas horas dedicadas a la asignatura. He intentado plasmarlo más que como apuntes como guía para el estudio. Es decir, si yo tuviera que volver a estudiar la asignatura lo haría siguiendo este orden. Al principio todo te parecerá soporífero, pero si tienes paciencia, quién sabe, a lo mejor al final hasta te resulta interesante. Por eso, es importante que al principio tengas un poco de paciencia y te empeñes en llegar por lo menos hasta la página 17 de esta guía. Si al llegar a este punto has entendido los ejemplos *(páginas 10 a 17)*, quizá valga la pena que vuelvas a empezar desde el principio, seguramente lo entenderás todo mucho mejor.

Que tengas mucha suerte.

José Manuel

PRELIMINARES

ESPECIFICACIÓN DE PROBLEMAS

- La especificación ha de ser "precisa" y "breve" por lo que se requiere una "especificación formal"

- Una técnica de especificación formal, basada en la lógica de predicados, se conoce como técnica "***pre/post***" :

  Si ***S*** *representa la función a especificar*
  Q *es un predicado que tiene como variables libres los parámetros de entrada de S*
  R *es un predicado que tiene como variable libres los parámetros de entrada y de salida de S*

  la especificación formal de S se expresa: ***{Q} S {R}***

- **Q** se denomina *PRECONDICIÓN* y caracteriza el conjunto de estados iniciales para los que está garantizado que **S** funciona (Se dice que la precondición describe las obligaciones del usuario del programa).

- **R** se denomina *POSTCONDICIÓN* y caracteriza la relación entre cada estado inicial y el estado final correspondiente. (Se dice que R describe las obligaciones del implementador).

- **{Q} S {R}** se lee como: *" Si S comienza su ejecución en un estado descrito por Q, S termina, y lo hace en un estado descrito por R ".*

  (Si el usuario llama al algoritmo en un estado no definido por Q, no es posible afirmar qué sucederá. La función puede no terminar, terminar con un resultado absurdo o hacerlo con un resultado razonable).

- Emplearemos la siguiente *notación* para **S**:

  a) Cuando las variables que aportan valores de entrada no pueden ser modificadas por el algoritmo:

  fun *nombre* (p_1:t_1;…;p_n:t_n) **dev** (q_1:r_1;…q_m:r_m)

  Denominamos ***parámetros*** al conjunto de las variables que forman la interface del algoritmo, distinguiendo entre parámetros de entrada (p_i) y parámetros de salida (q_j)

  Las t_i representan los tipos de datos asociados a los *parámetros de entrada*

  Las r_j son los tipos de datos asociados a los *parámetros de salida*

Como hemos comentado, la notación **fun** se empleará siempre que los parámetros de entrada y de salida formen conjuntos disjuntos.

b) Cuando un algoritmo puede modificar una variable, la notación será:

accion *nombre* (p_1:cf t_1;…;p_n:cf t_n)

donde **cf** puede ser:

- **ent** si el parámetro es solo de entrada
- **sal** si el parámetro es solo de salida
- **ent/sal** si es un parámetro de entrada/salida

y t_i representa los tipos de datos asociados a los parámetros.

EJEMPLOS Y EJERCICIOS:

Del ***Texto Base*** conviene entender los ejemplos: "Algoritmo que calcule el cociente por defecto y el resto de la división entera"; "Máximo de un vector"; "Posición del primer máximo del vector" y ¿Cómo especificar la moda?

De la ***colección de problemas curso 1999-2000*** conviene hacer los ejercicios **1.1.2** (Hallar precondiciones); **1.3.1** (Especificación de funciones); **6.1** (Algoritmo que calcule cuántos elementos de un vector son múltiplos de un natural k); **6.2** (Algoritmo que resuelve un sistema de ecuaciones lineales); **6.3** (La celebridad); **6.4** (Módulo de un vector); **6.5** (media y moda); **6.6** (Posición del primer máximo de un vector); **6.7** (Producto escalar de dos vectores); **6.8** (desplazamiento circular de los elementos de un vector); **6.9** (Descomposición de un cuadrado) y **6.10** (longitud del menor segmento de un vector que contenga, al menos, k ceros)

LÓGICA DE PREDICADOS

Aunque se estudia mejor en la asignatura Lógica Matemática conviene estudiar con detenimiento, sobre todo de cara al test, algunas definiciones como:

- Variables libres y ligadas
- Satisfactibilidad, consecuencia lógica y equivalencia
- **Especificación con predicados y Convenios sobre cuantificadores y sustituciones**

EJEMPLOS Y EJERCICIOS:

De la ***colección de problemas curso 1999-2000*** conviene hacer los ejercicios **1.1.1** (Expresar mediante la lógica de predicados una serie de enunciados en lenguaje natural); **1.2.1** (Extensión de un conjunto de estados); **1.2.2** (Intensión de un conjunto de estados); **1.2.3** (Potencia); **1.2.4** (Formalización de enunciados) y **1.2.5** (Formalización de enunciados)

DISEÑO RECURSIVO

CONEPTOS BÁSICOS, TERMINOLOGÍA Y NOTACIÓN

- La recursividad permite que un procedimiento o función hagan referencia a sí mismos dentro de su definición.

  - Ello conlleva que una invocación al programa recursivo genera una o más llamadas al propio subprograma, cada una de las cuales genera nuevas invocaciones y así sucesivamente.
  - Si la definición está bien hecha, las cadenas de invocaciones así formadas terminan felizmente en alguna llamada que no genera nuevas invocaciones
  - Ejemplo: Cálculo de la potencia n-ésima de un número “a”

    Como quiera que $a^n = a^{n-1} * a$, podemos definir la función **potencia** de la siguiente forma: **potencia**(a,n) = **potencia**(a,n-1)***a**

    Para calcular $\mathbf{a^n}$ la función **potencia** calcula primero $\mathbf{a^{n-1}}$ mediante la llamada recursiva a la función con los parámetros **(a,n-1)** y el resultado de la llamada recursiva se multiplica por **a** para calcular el resultado de la función. Pero, para calcular **potencia**(a,n-1) se produce una nueva llamada recursiva: **potencia**(a,n-1)=**potencia**(a,n-2)***a** ; por lo que para saber lo que vale $\mathbf{a^{n-1}}$ previamente la función deberá calcular el valor de $\mathbf{a^{n-2}}$ y multiplicar el resultado obtenido por **a**.

    Pero el proceso no se puede repetir indefinidamente. Debe existir alguna llamada que no genere nuevas llamadas recursivas. En este ejemplo como $a^0=1$ cuando tengamos que calcular **potencia**(a,0) en lugar de realizar una nueva llamada recursiva, indicaremos a la función que directamente devuelva el valor **1**.

    Veamos cómo se calcularía 5^3:

    potencia(5,3) = **potencia**(5,2) *** 5**
    potencia(5,2) = **potencia**(5,1) *** 5**
    potencia(5,1) = **potencia**(5,0) *** 5**
    potencia(5,0) = **1**

    potencia(5,3) = [(1*5)*5]*5 = 125

- El esfuerzo de razonamiento es menor trabajando en recursivo que en iterativo

  - Veamos mediante el ejemplo de la potencia n-ésima la diferencia de razonamientos recursivo e iterativo
  - *ITERATIVO*: el problema **P** sería calcular $\mathbf{a^n}$

    Para resolverlo optamos por multiplicar n veces a por a por lo que transformamos el problema original **P** en otro “***p***” que consiste en multiplicar dos números (x,a)

"x" irá acumulando los productos de *a-es*, por lo que se inicia a 1 y a cada iteración su nuevo valor se obtendrá mediante la asignación **x := x*a**

Repitiendo exactamente n veces el problema "***p***" se consigue solucionar el problema original **P**

- *RECURSIVO*: tratamos de resolver el problema **P** suponiendo que **P** ya está resuelto para otros datos del mismo tipo pero en algún sentido *más sencillos*; es decir, suponiendo que ya sabemos calcular a^{n-1} resolver el problema original a^n es inmediato.

  En el razonamiento recursivo hay involucrado un solo problema **P** y un solo tipo de datos

- Para que el razonamiento recursivo sea correcto se requiere que la sucesión de invocaciones no sea infinita. Se llegará entonces a unos datos lo suficientemente sencillo como para que el problema quede resuelto directamente. Diremos que se trata de un ***caso trivial***

  En nuestro ejemplo *n=0 es un caso trivial* → $a^n = 1$

ASPECTO DE UN PROGRAMA RECURSIVO

- Habrá una o más instrucciones condicionales dedicadas a separar los tratamientos correspondientes al caso (casos) trivial(es) de los correspondientes al caso (o casos) no trivial(es).

- El tratamiento de los casos no triviales será:

1.- Se calculará el "*sucesor*" (dato que reduce el tamaño del problema)→"*subproblema*"

2.- A continuación se produce una llamada recursiva para obtener la solución al *subproblema*

3.- Finalmente se opera sobre el resultado obtenido para el *subproblema* a fin de calcular la solución para el *problema inicial*

En nuestro ejemplo: **potencia**(a,n) es el problema

Caso trivial: *n=0* → **potencia**(a,n)=**1**

Caso no trivial: *n>0*

1.- *sucesor*: n-1

2.- *llamada recursiva*: **potencia**(a,n-1)

3.- como potencia(a,n)=a^n y potencia(a,n-1)=a^{n-1} tendremos potencia(a,n)=potencia(a,n-1)*a

luego para *n>0* → **potencia**(a,n)=**potencia**(a,n-1)***a**

SINTAXIS

- En el LR (lenguaje recursivo) no hay instrucciones en el sentido d órdenes que han de ser seguidas secuencialmente por un computador. En su lugar hay **expresiones**

- La instrucción "***condicional***" de los lenguajes imperativos se convierte aquí en una expresión.
  Un condicional está formado por un conjunto de alternativas de la forma **Bi → Ei** separadas entre sí por el símbolo **[]**

- La ejecución de un *condicional* comienza evaluando ***todas*** las expresiones booleanas **Bi**. Si *ninguna*, *o más de una*, resultan ciertas, el resultado del condicional es *indefinido* y el programa aborta su ejecución.

- Veamos pues el "*aspecto*" del programa que calcula la potencia n-ésima de un número escrito en LR:

```
fun potencia(a:entero; n:natural) dev (p:entero)

        caso  n=0  →   1
         []   n>0  →   a*potencia(a,n-1)
        fcaso

ffun.
```

- En ocasiones una expresión **E** ha de aparecer repetidas veces como subexpresión de otra expresión más compleja **E'** . Para evitar la escritura repetida y evitar tener que evaluar dicha expresión más de una vez, se introduce una declaración local. La sintaxis es ***"sea x = E en E'(x) "*** e indica que la expresión **E** sólo se evalúa una vez, que llamamos **x** al resultado de su evaluación y que dicho valor puede aparecer repetidas veces en **E'**

Ejemplo: Supongamos que al escribir un programa en LR tenemos:

```
caso  x<a+b → …        Haríamos:   Sea s=a+b en
 []   x=a+b → …                    caso  x<s  →  …
 []   x>a+b → …                     []   x=s  →  …
                                    []   x>s  →  …
```

TERMINOLOGÍA:

- Cuando una función recursiva genera a lo sumo una llamada interna por cada llamada externa, diremos que es una función ***recursiva lineal*** (recursiva *simple*)

- Cuando genera dos o más llamadas internas por cada llamada externa diremos que es ***recursiva no lineal*** (recursiva *múltiple*).

- Podría ocurrir que en el texto de una función recursiva lineal aparecieran varias llamadas recursivas, una en cada alternativa diferente de una instrucción condicional. La recursividad sigue siendo lineal.

ESQUEMA DE UNA FUNCIÓN RECURSIVA LINEAL

Afortunadamente en los ejercicios de este curso todas las funciones con las que trabajaremos son recursivas lineales. (Ojo para el test puede caer algo de recursividad múltiple).

El esquema general es el siguiente:

$\{Q(\overline{x})\}$

fun $f(\overline{x} : T_1)$ **dev** $(\overline{y} : T_2)$

  caso $B_t(\overline{x}) \rightarrow triv(\overline{x})$

  [] $B_{nt}(\overline{x}) \rightarrow c(f(s(\overline{x})), \overline{x})$

  fcaso

ffun

$\{R(\overline{x}, \overline{y})$

Donde:**1**.- Los parámetros $\overline{x}$ e $\overline{y}$ han de entenderse como tuplas "$x_1,\ldots, x_n$" e "$y_1,\ldots,y_n$" *(parámetros de entrada y resultados respectivamente)*

2.- $\mathbf{B_t(x)}$ representan los casos triviales (*protección caso trivial*); y $\mathbf{B_{nt}(x)}$ los casos no triviales (*protección caso no trivial*)

Para todo (x) que satisfaga la precondición debe ser $B_t(x) \wedge B_{nt}(x) \equiv falso$; es decir los casos triviales y no triviales han de ser conjuntos ***disjuntos***.

3.- **triv(x)** representa el *resultado de la función para el caso trivial*

4.- Vallamos más despacio al analizar la expresión **c(f(s(x)),x)**.

Volvamos al ejemplo de la potencia n-ésima.

En este caso tenemos:

$(x) \equiv \{a,n\}$; $(y) \equiv \{p\}$; $f(x) \equiv potencia(a,n)$

$Bt(x) \equiv n=0$ y $triv(x) \equiv 1$

$Bnt(x) \equiv n>0$ y c(f(s(x)),x) ≡ ¿?

s(x) es la función "***sucesor***" que en nuestro ejemplo era $s(x) \equiv n\text{-}1$.

f(s(x)) es la llamada recursiva a la función, pero con el parámetro sucesor. En el ejemplo $f(s(x)) \equiv potencia(a,n\text{-}1)$.

c(f(s(x)),x) es la función "***combinar***". Para resolver el problema original (a^n) a partir del resultado devuelto por la llamada recursiva (a^{n-1}) es preciso realizar una operación adicional que llamamos "*combinar*" (se combina el resultado devuelto con parte de los parámetros originales). En este caso, como $a^n = a * a^{n-1}$ se puede concluir que potencia(a,n) = a * potencia(a,n-1).

Es decir; $\underline{c(f(s(x)),x) \equiv a * potencia(a,n\text{-}1)}$

5.- Finalmente **R(x,y)** es la ***postcondición*** de la función **f** y establece la relación que ha de cumplirse entre los parámetros (x) y los resultados (y). [$p = a^n$]

- Cuando la función "*combinar*" no es necesaria, es decir *f(x) = f(s(x))* en el caso no trivial, diremos que f es recursiva ***final***. En caso contrario, f es recursiva ***no final***.
- Las funciones recursivas finales son más eficientes en cuanto a tiempo y en orden de complejidad y su transformación a iterativo es más directa.

ANALISIS POR CASOS Y COMPOSICIÓN

No todos los problemas son tan sencillos como el ejemplo de la potencia n-ésima

En esta fase tendremos que estudiar cómo se pueden descomponer recursivamente los datos de problema.

No hay reglas ya que se trata de la etapa más creativa del diseño.

- Hay que tener en cuenta que en general habrá más de una forma de hacerlo. La solución más eficiente se alcanza casi siempre escogiendo la descomposición recursiva que más drásticamente reduce el tamaño del problema.
- En esta etapa se analizan los casos que se pueden presentar a la función, identificando los que pueden considerarse *no triviales*, y qué solución hay que aplicar en estos casos, y bajo qué condiciones el problema ha de considerarse *trivial* y cómo se resuelve entonces.
- Aunque después se estudiará cómo verificar formalmente la corrección del diseño, hay dos comprobaciones importantes que hacer en esta etapa:

  1.- Asegurarse que la reducción aplicada al problema en el caso no trivial conduce a problemas cada vez más pequeños, (reducción del tamaño del problema), que necesariamente terminan en el caso trivial.

  2.- Asegurarse de que entre los casos triviales y no triviales se han cubierto todos los estados previstos por la postcondición.

- La etapa de **composición** consiste solo en expresar, en forma de programa *LR*, el resultado del **análisis por casos.** Hay dos consideraciones importantes:

  1.- Asegurarse de que las condiciones de las instrucciones condicionales son mutuamente excluyentes.

  2.- Introducir declaraciones locales para evitar calcular repetidas veces la misma expresión

EJEMPLOS Y EJERCICIOS:

Llegados a este punto es conveniente leer y entender los siguientes ejemplos del ***Texto Base***:

Algoritmo de Euclides para calcular el mcd de dos números.
La división entera
La potencia entera reconsiderada.

TÉCNICAS DE INMERSIÓN

- Sucede con frecuencia que no es posible abordar directamente el diseño recursivo de una función **f**.
- Una técnica que puede solucionar el diseño consiste en: "Definir una función **g**, más general que **f**, es decir, con más parámetros y/o resultados que, para ciertos valores de los nuevos parámetros calcula lo mismo que **f**"
- Diremos que hemos aplicado una inmersión de f en g

  g → *función Inmersora* ***f*** → *función Sumergida*

  generalmente usaremos para **g** con el mismo nombre que **f** precedido por una "*i*"
- Para calcular la función original basta con establecer el valor inicial de los nuevos parámetros que hacen que la función inmersora se comporte como la sumergida.

¿Qué quiere decir todo esto?.

Bueno, si has tenido paciencia y has llegado hasta este punto, creo que lo que viene a continuación puede aclararte bastante las ideas.

Vamos a ver un par de ejemplos.

El primero lo he tomado prestado del profesor Jerónimo Quesada a quien quiero agradecer públicamente la inestimable labor que realiza ya que sin sus ejercicios jamás habría llegado a entender de qué va esta asignatura, y cuya página web es visita obligada para cualquier estudiante de programación2. (*http://www.bitabit.com*)

El segundo corresponde a la práctica del curso 99-00 (la que me ha tocado en suerte).

A partir de este punto se hacen referencias continuamente a estos dos ejemplos a medida que avancemos en la teoría.

EJEMPLO1: Dado un vector de enteros de dimensión N diseñar un algoritmo que calcule la suma de los elementos del vector.

ESPECIFICACIÓN: $\{Q \equiv \text{cierto}\}$
fun suma(v: vector[1..N] de enteros) dev (s: entero)
$\{R \equiv s = \sum \alpha \in \{1..N\}.\ v[\alpha]\}$

(Confío que no tengas problema en escribir la especificación por ti mismo.)

DISEÑO RECURSIVO:

No parece fácil a primera vista encontrar una solución recursiva para esta función. Será preciso realizar una inmersión.

¿Qué es una inmersión?. Básicamente consiste en especificar un algoritmo en el que o bien se ha <u>*debilitado la postcondición*</u> (lo que dará lugar a una inmersión ***no-final***) o bien se ha <u>*fortalecido la precondición*</u> (lo que dará lugar a una inmersión de tipo ***final***)

- Para obtener inmersiones la técnica más adecuada consiste en "<u>*manipular la postcondición*</u>" del algoritmo original. Consiste en <u>*introducir variables en lugar de constantes*</u>.

  En este ejemplo veamos qué ocurre si introducimos una variable "i" en sustitución de la constante "N"

  La postcondición original es: $R \equiv s = \sum \alpha \in \{1..N\}.\ v[\alpha]$ y al introducir la manipulación propuesta se obtiene $R1 \equiv s' = \sum \alpha \in \{1..i\}.\ v[\alpha]$

  Para obtener R a partir de R1 tendríamos que hacer i = N, luego se puede escribir:

  $R \equiv (\sum \alpha \in \{1..i\}.\ v[\alpha]) \wedge (i = N)$; es decir:

  $R \equiv R1 \wedge R2$ con <u>$R1 \equiv \sum \alpha \in \{1..i\}.\ v[\alpha]$</u> y <u>$R2 \equiv i = N$</u>

  R1 es *más débil* que *R* en el sentido de que "pide menos" a la función. Mientras la postcondición original pedía a "s" que devolviera la suma de "todos" los elementos del vector, la nueva postcondición R1 "solo" exige que "s'" contenga la suma de los i-primeros elementos.

  Como hemos introducido un nuevo parámetro, la nueva precondición ha de incluir las condiciones que debe cumplir la variable incorporada. En este caso está claro que $1 <= i <= N$ al tratarse de un índice que debe ser válido para el vector.

  Pero hay que tener en cuenta que se ha de intentar conseguir la ***precondición más débil***. Por eso nos planteamos ampliar el rango de valores de i. ¿Qué ocurre si i=0?. En ese caso *s =* **åaÎ** *{1..0}. v[***a***]* que, según convenio sobre cuantificadores, (*ver Peña*), sabemos que da como resultado $s=0$ al anularse el rango del sumatorio.

  Por tanto, podemos plantear $0<=i<=N$ que además nos "regala" una solución trivial al algoritmo: para $i=0 \rightarrow s=0$

Pero, vayamos un poco más despacio. En primer lugar escribamos la ***especificación*** de la función inmersora *isuma*:

> *{Q ≡ 0<=i<=N}*
> *fun isuma(v: vector[1..N] de enteros; i: natural) dev (s: entero)*
> *{R ≡ s =* $\Sigma \alpha \in$ *{1..i}. v[*α*]}*

ANÁLISIS POR CASOS: (*Trabajamos sobre valores de la nueva variable introducida*)

Como ya hemos comentado para i=0 → s=0 (*caso trivial*)
¿ y para i>0 ? ¿? (*caso no trivial*)

Para resolver el caso no trivial seguimos los siguientes pasos:

a) Realizamos una llamada recursiva a la función pero con el tamaño del problema reducido: *isuma(v, i-1)*

b) El resultado de esa llamada debe cumplir la postcondición y por tanto, si llamamos *s'* al resultado de *isuma(v, i-1)* tendremos *s' =* $\Sigma \alpha \in$ *{1..i-1}. v[*α*]*

c) Realizamos las operaciones adicionales necesarias para obtener un resultado que cumpla la postcondición para los parámetros originales a partir del valor devuelto por la llamada recursiva:

Comparando el resultado de la llamada recursiva (*s'*) y el de la llamada con los parámetros originales (*s*)

s = v[1] + v[2] + ...+ v[i-1] + v[i] *s'* = v[1] + v[2] + …. + v[i-1]

observamos que *s = s' + v[i]*

En definitiva *isuma(v,i) = isuma(v,i-1) + v[i]*

COMPOSICIÓN: El algoritmo completo correspondiente a la función *isuma* será:

```
{Q ≡ 0<=i<=N}
   fun isuma(v: vector[1..N] de enteros; i: natural) dev (s: entero)
        caso   i=0   →   0
         []    i>0   →   isuma(v, i-1) + v[i]
        fcaso
   ffun
{R ≡ s = Σ α∈{1..i}. v[α]}
```

LLAMADA INICIAL: Finalmente, pero no menos importante, es saber bajo qué condiciones la nueva función *isuma* resuelve el problema original (*llamada inicial*). Es decir ¿qué parámetros hacen que la función *isuma* calcule la suma de los *N* elementos del vector? (éste era el problema original). En este caso es evidente que cuando *i = N*, la función isuma realizará el cálculo deseado.

Por tanto *suma(v) = isuma(v,N)*

La función obtenida es recursiva ***no-final***; es decir, tras la llamada recursiva es necesario realizar una operación adicional antes de devolver el resultado *[c(f(s(x)),x)]*

DISEÑO RECURSIVO FINAL: *¿Qué hubiera ocurrido si en lugar de "debilitar" la postcondición la mantenemos constante y lo que hacemos es "reforzar" la precondición?*

- En la precondición se pedirá que parte del trabajo "*se dé hecho*".

En nuestro ejemplo podemos pedir en la precondición que se haya calculado previamente la suma de los *i-primeros* elementos. Estas sumas parciales se almacenan en un nuevo parámetro que llamamos *parámetro acumulador*.

Así las cosas, nuestro algoritmo solo deberá preocuparse de "*acumular*" a dicho parámetro la suma de los elementos $v[i+1]...v[n]$ del vector, ya que a la entrada al algoritmo el parámetro acumulador debe tener ya la suma de los elementos $a[1]...a[i]$.

En definitiva, partimos, igual que antes de:

$$R \equiv R1 \wedge R2 \quad con \quad \underline{R1 \equiv \sum \alpha \in \{1..i\}.\ v[\alpha]} \quad y \quad \underline{R2 \equiv i = N}$$

Pero con la diferencia de que ahora <u>*mantenemos como postcondición R*</u> y lo que hacemos es introducir un <u>parámetro adicional</u>, $r = \sum \alpha \in \{1..i\}.\ v[\alpha]$ (*acumulador*).

Pongamos entonces que la nueva precondición es:

$$\{Q \equiv 0<=i<=n \ \wedge \ r = \sum \alpha \in \{1..i\}.\ v[\alpha]\}$$

(Observa que se han introducido dos parámetros adicionales i, r y que a r se le pide, a la entrada del algoritmo, que contenga la suma de los i primeros elementos del vector).

Pues bien, la especificación de la nueva función *isuma2* será:

$\{Q \equiv 0<=i<=n \ \wedge \ r = \sum \alpha \in \{1..i\}.\ v[\alpha]\}$
fun isuma2(v:vect[1..N] de entero; i:nat; r:entero) dev s:entero
$\{R \equiv s = \sum \alpha \in \{1..n\}.\ v[\alpha]\}$

¿Cómo utilizar esta nueva función *isuma2* para resolver el problema original?. Es decir, ¿cómo hacer que *isuma2* devuelva la suma de todos los elementos del vector?

Parece obvio que si $i=0$, entonces $r=0$, es decir que al llamar a la función con el valor del parámetro i=0 "se le pasa" la suma de 0 elementos y por tanto el trabajo a realizar por la función será la suma de todos los elementos.

Esto es *suma(v) = isuma2(v,0,0)* será la <u>*llamada inicial*</u>.

ANÁLISIS POR CASOS:

- <u>*Casos triviales*</u>: Si la llamada se hace con $i=n$ resultará que *r* ya tiene la suma de los **n** elementos del vector ($r = \sum \alpha \in \{1..i\}.\ v[\alpha]$) y por tanto la función se limitará a <u>*devolver el valor de r.*</u>

¿ Y cuando i<n? ¿cómo resuelve la función los casos no triviales?

Como sabemos que la función que vamos a obtener a través de esta inmersión será recursiva final, no hará operaciones adicionales y podemos plantear que la fución planteará la solución al caso no trivial como una llamada recursiva a la función; es decir:

$i<n \rightarrow isuma2(v, i',r')$

Puesto que el caso trivial se da para $i=n$, para que la nueva llamada nos acerque al caso trivial podemos plantear $i' = i+1$

Entonces, en la llamada *isuma2(v, i+1, r')*, r' debe contener la suma de los $i+1$ primeros elementos del vector ya que debe cumplirse la ***precondición*** , es decir debe ser $r' = \Sigma \alpha \in \{1..i+1\}.\ v[\alpha]\}$

Como r contenía la suma de los *i-primeros* elementos, podemos concluir que $r' = r+v[i+1]$, por tanto la llamada interna para el caso no trivial será:

$$isuma2(v,\ i+1,\ r+v[i+1])$$

COMPOSICIÓN: El algoritmo completo correspondiente a la función *isuma2* será:

```
{Q ≡ 0<=i<=n  ^  r = Σ α ∈ {1..i}. v[α]}
fun isuma2(v:vect[1..N] de entero; i:nat; r:entero) dev s:entero
        caso   i=n  →  r
         []    i>0  →     isuma2(v, i+1,r+v[i+1])
        fcaso
ffun
{R ≡ s = Σ α ∈ {1..n}. v[α]}
```

obteniendo de esta forma una función recursiva final.

No debemos caer en el error de pensar que estas soluciones son únicas. Más bien todo lo contrario.

Por ejemplo, si a la hora de debilitar la postcondición, nos hubiéramos planteado:

$R \equiv R1 \wedge R2$ *con* $R1 \equiv s = \Sigma \alpha \in \{i+1..N\}.\ v[\alpha]$ *y* $R2 \equiv i=0$

todo habría sido diferente:

La inmersión no final sería:

```
{Q ≡ 0<=i<=N}
   fun isuma(v: vector[1..N] de ent; i: nat) dev (s: ent)
         caso     i=N  →  0
          []      i<N  →  isuma(v, i+1) + v[i+1]
         fcaso
   ffun
{R ≡ s = Σ α ∈ {i+1..N}. v[α]}

llamada inicial:   suma(v) = isuma(v,0)
```

y la inmersión final sería:

```
{Q ≡ 0<=i<=n  ^  r = Σ α ∈ {i+1..N}. v[α]}
fun isuma2(v:vect[1..N] de ent; i:nat; r:ent) dev s:ent
         caso     i=0  →  r
          []      i>0  →  isuma2(v, i-1,r+v[i])
         fcaso
ffun
         {R ≡ s = Σ α ∈ {1..N}. v[α]}

llamada inicial:  suma(v) = isuma2(v ,N,0)
```

EJEMPLO2: En el conocido juego del Tetris se dispone de una pantalla por donde caen las piezas que se codifica en una matriz de n filas por m columnas definida como sigue:

t: vector[1..n, 1..m] de nat

Siendo la primera coordenada de la matriz el número de la fila y la segunda coordenada el número de columna.

Cada posición almacena un número natural que representa el código del color de la pieza que ocupa dicha casilla. El valor 0 representa una casilla vacía.

Se pide diseñar una función que dada una matriz t con las características descritas y un número de fila f entre 1 y n nos diga si la fila f-ésima está completamente llena de piezas y, por tanto, representa una línea completada en el juego.

ESPECIFICACIÓN:

$\{Q \equiv 1<=f<=n\}$
fun linea(t: vector[1..n, 1..m] de nat; f: nat) dev (b: bool)
$\{R \equiv b = \forall \alpha \in \{1..m\}.\ t[f,\alpha]>0\}$

DISEÑO RECURSIVO: INMERSIÓN NO FINAL

$R \equiv R1 \wedge R2$ con $R1 \equiv (b = \forall \alpha \in \{1..i\}.\ t[f,\alpha]>0)$ y $R2 \equiv (i = m)$

Razonando de manera análoga a como se hizo en el ejemplo1 se obtiene el código:

```
{Q ≡ (1<=f<=n) ^ (0<=i<=m)}
fun ilinea(t: vector[1..n, 1..m] de nat; f,i: nat) dev (b: bool)
        caso   i=0  →  cierto
         []    i>0  →  ilinea(t, f, i-1)  ^  (t[f,i]>0)
        fcaso
ffun
{R ≡ b = ∀α ∈ {1..i}. t[f,α]>0}
```

y la llamada inicial será: <u>linea(t, f) = ilinea(t, f, m)</u>

Notas: 1.- Aunque inicialmente se piense en tomar 1<=i<=m se puede ampliar el rango con el valor i=0 ya que para dicho valor b = ∀α ∈ {1..0}. t[f,α]>0 ≡ "cierto" por convenio sobre cuantificadores. Además proporciona la solución para el caso trivial.

2.- Comparando ilinea(t, f, i) = t[f,1]>0 ^ t[f,2]>0 ^ … ^ t[f,i-1]>0 ^ t[f,i]>0
con ilinea(t, f, i-1) = t[f,1]>0 ^ t[f,2]>0 ^ … ^ t[f,i-1]>0
se llega a la conclusión: ilinea(t, f, i) = ilinea(t, f, i-1) ^ (t[f,i]>0)

3.- Cuando i=m la función ilinea comprobará que todos los elementos de la fila f-ésima son distintos de cero (línea completada); y por tanto calcula lo mismo que la función original linea. De ahí que la llamada inicial sea:

linea(t, f) = ilinea(t, f, m)

DISEÑO RECURSIVO FINAL

$\{Q \equiv (1<=f<=n) \wedge (0<=i<=m) \wedge (bb = \forall \alpha \in \{1..i\}.\ t[f,\alpha]>0)\}$

```
fun ilinea2(t: vector[1..n, 1..m] de nat; f,i: nat; bb: bool) dev (b: bool)
        caso   i=m  → bb
         []    i<m  → ilinea2(t, f, i+1, bb ^ t[f,i+1]>0)
        fcaso
ffun
```

$\{R \equiv b = \forall \alpha \in \{1..m\}.\ t[f,\alpha]>0\}$

y la llamada inicial será: <u>linea(t, f) = ilinea2(t, f, 0, cierto)</u>

Notas: El razonamiento seguido es similar al del ejemplo1 con las particularidades siguientes:

1.- El parámetro *bb* comprueba si los i-primeros elementos de la fila f-ésima están o no ocupados. Cuando i=m al invocar la función el parámetro bb ya tendrá el valor de verdad correspondiente a la comprobación de todos los elementos de la fila f-ésima, por lo que la función devolverá el valor de bb. (caso trivial)

2.- En el caso no trivial, cuando i<m, y, dado que el caso trivial se da para i=m, lo lógico es pensar que el parámetro i se incremente (i' = i+1), con lo que la llamada interna será ilinea2(t, f, i+1, bb'). Esta llamada se hará con parámetros tales que se cumpla la precondición, y por tanto:

$bb' = \forall \alpha \in \{1..i+1\}.\ t[f,\alpha]>0 \quad \equiv \quad bb' = t[f,1]>0 \wedge \ldots \wedge t[f,i]>0 \wedge t[f,i+1]>0$
como $bb = \forall \alpha \in \{1..i\}.\ t[f,\alpha]>0 \quad \equiv \quad bb = t[f,1]>0 \wedge \ldots \wedge t[f,i]>0$
de la comparación de ambas se concluye que $bb' = bb \wedge t[f,i+1]>0$
(solución para el caso no trivial)

3.- Respecto a la llamada inicial es obvio que cuando *i=0* → *bb = "cierto"* y la función *ilinea2* tendrá que comprobar todos y cada uno de los elementos de la fila f-ésima, con lo cual *"realizará el mismo trabajo"* que la función *linea*.

TEORIA, EJEMPLOS Y EJERCICIOS:

Llegados a este punto es conveniente intentar entender las explicaciones teóricas de las técnicas de inmersión dadas en el ***Texto Base***, intentando relacionar la teoría con lo puesto de manifiesto en estos ejemplos.

Sería conveniente hacer el siguiente ejercicio:

EJEMPLO3: Queremos una función que dados dos vectores a y b, cuyo tipo de datos es
Tipovect = vector[1..n] de enteros (siendo n una constante)
Calcule el producto escalar de a y b

EJEMPLO3: SOLUCIÓN

ESPECIFICACIÓN:

$\{Q \equiv \text{cierto}\}$
fun prodesc(a,b: vector[1..n] de ent) dev (p: ent)
$\{R \equiv p = \sum \alpha \in \{1..n\}.\ a[\alpha] * b[\alpha]\}$

DISEÑO RECURSIVO: INMERSIÓN NO FINAL

$R \equiv R1 \wedge R2$ *con* $R1 \equiv (p = \sum \alpha \in \{j..n\}.\ a[\alpha] * b[\alpha])$ *y* $R2 \equiv (j = 1)$

```
{Q ≡ (1<=j<=n)}
fun iprodesc(a,b: vector[1..n] de ent; j:nat) dev (p: ent)
        caso   j=n+1  →  0
         []    j<n+1  →  iprodesc(a, b, j+1)  +  a[j]*b[j]
        fcaso
ffun
{R ≡ p = Σα ∈ {j..n}. a[α] * b[α]}
```

y la llamada inicial será: prodesc(a,b) = iprodesc(a,b,1)

DISEÑO RECURSIVO FINAL

```
{Q ≡ (1<=j<=n)  ^  s = Σα ∈ {j..n}. a[α] * b[α]}
fun iprodesc2(a,b: vector[1..n] de ent; j: nat; s: ent) dev (p:ent)
        caso   j=1  →  s
         []    j>1  →  iprodesc2(a, b, j-1, s+a[j-1]*b[j-1])
        fcaso
ffun
{R ≡ P = Σα ∈ {1..n}. a[α] * b[α]}
```

y la llamada inicial será: prodesc(a, b) = iprodesc2(a, b, n+1, 0)

TÉCNICAS DE DESPLEGADO Y PLEGADO

Convierte programas *recursivos no finales en recursivos finales*.

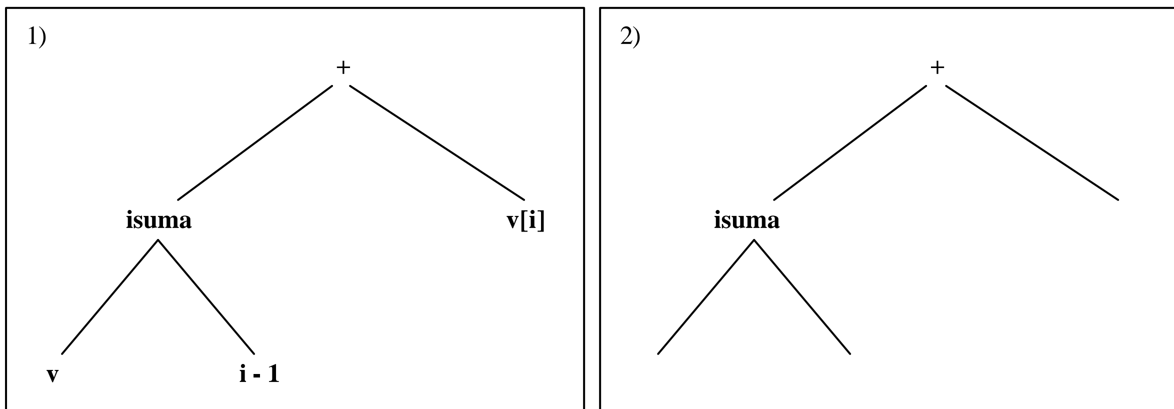
Se trata de transformar f, recursiva no final, en otra función g, recursiva final que calcule lo mismo que f.

Los pasos a dar son los siguientes:

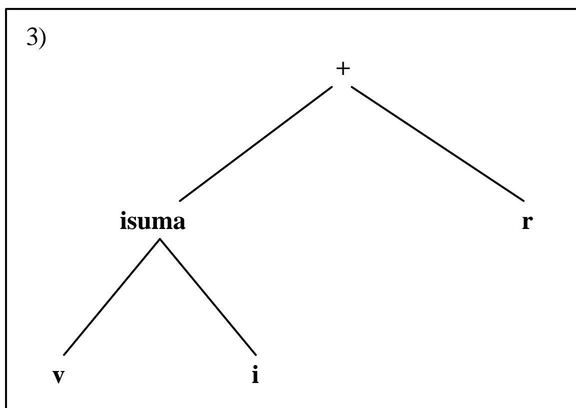
1.- Generalización: Se define una función $g(\vec{x}, \vec{w})$ como una expresión que depende de f y representa una generalización de la misma y se establece un valor inicial w_{ini} para el cual g se comporta como f

- En la práctica se construye el *árbol sintáctico* de la operación ***c*** (*combinar*). A continuación se conserva el camino que va desde la raíz de árbol hasta la invocación a f y cada subárbol de éste camino se sustituye por un parámetro de inmersión diferente.
- Si la función ***c*** tiene elemento neutro w_0, entonces g es una generalización de f que se comporta igual que ella para el valor inicial $w = w_0$.

Volvamos a retomar nuestro ejemplo1 y dibujemos el árbol sintáctico de la operación ***c***



Las ramas laterales se rellenan con parámetros de inmersión y la ramas que parten de *isuma* se rellenan con los parámetros originales:



La nueva función, que llamaremos ***iisuma*** será:

$$iisuma(v, i, r) = isuma(v, i) + r$$

Como el elemento neutro de la suma es el cero, tendremos:

$$iisuma(v, i, 0) = isuma(v, i)$$

2.- Desplegado:

Para poder aplicar el desplegado y posterior plegado, la función ***c*** *(combinar) ha de poseer* ***elemento neutro*** *y ser* ***asociativa***

2.1.- Partiendo de la igualdad ***iisuma(v, i, r) = isuma(v, i) + r*** , sustituimos la función isuma por su desarrollo:

$$\text{iisuma(v, i, r)} = \begin{Bmatrix} \text{caso } i = 0 \rightarrow 0 \\ \bullet \quad i > 0 \rightarrow \text{isuma(v, i - 1)} + v[i] \end{Bmatrix} + r$$

2.2.- Trasladamos la operación adicional a cada uno de los casos *(desplegado)*

$$\text{iisuma(v, i, r)} = \begin{Bmatrix} \text{caso } i = 0 \rightarrow 0 + r \\ \bullet \quad i > 0 \rightarrow \text{isuma(v, i - 1)} + v[i] + r \end{Bmatrix}$$

2.3.- Aplicando las propiedades, elemento neutro al caso trivial y asociativa al no trivial

$$\text{iisuma(v, i, r)} = \begin{Bmatrix} \text{caso } i = 0 \rightarrow r \\ \bullet \quad i > 0 \rightarrow \text{isuma(v, i - 1)} + (v[i] + r) \end{Bmatrix}$$

3.- La expresión del caso no trivial de iisuma tiene el mismo aspecto de su definición; es decir, es igual a la función isuma, (con unos determinados parámetros), mas un sumando adicional (v[i] + r).

Es decir isuma( ) + ( ) tiene el aspecto de isuma(v, k) + p = isuma(v, k, p); por tanto, si hacemos k=i-1 y p=v[i]+r tenemos:

isuma(v, i-1) + (v[i] + r) = iisuma(v, i-1, v[i]+r)

es decir:

$$\text{iisuma(v, i, r)} = \begin{Bmatrix} \text{caso } i = 0 \rightarrow r \\ \bullet \quad i > 0 \rightarrow \text{iisuma(v, i - 1, } v[i] + r) \end{Bmatrix}$$

que resulta ser recursiva final (se ha eliminado la función combinar).

La función recursiva final obtenida será:

```
fun iisuma(v : vect[1..n]de ent; i : nat, r : ent) dev (s : ent)
    caso i = 0 → r
     •    i > 0 → iisuma(v, i - 1, v[i] + r)
ffun
```

La llamada inicial de esta función, es decir, aquella que calcula lo mismo que la función original suma será: ***iisuma(v, N, 0) = isuma(v, N) = suma(v)***

La postcondición de esta función será la misma que la de la función isuma para el caso i=N; es decir la original de la función suma: $R \equiv s = \sum \alpha \in \{1..N\}.\ v[\alpha]$

La precondición deberá incluir condiciones adecuadas para el nuevo parámetro "*r*". Estas condiciones se pueden deducir del hecho de que ***iisuma = isuma(v, i) + r***

la función ***iisuma(v, i, r)*** devuelve $\sum \alpha \in \{1..N\}.\ v[\alpha]$
y la función ***isuma(v, i)*** devuelve $\sum \alpha \in \{1..i\}.\ v[\alpha]$

luego: ***iisuma = isuma(v, i) + r*** $\rightarrow \quad \sum \alpha \in \{1..N\}.\ v[\alpha] = \sum \alpha \in \{1..i\}.\ v[\alpha] + r$,

por tanto, podemos concluir que $r = \sum \alpha \in \{i+1..N\}.\ v[\alpha]$

La precondición será: $\{Q \equiv (0<=i<=N) \wedge (r = \sum \alpha \in \{i+1..N\}.\ v[\alpha])\}$

Para finalizar reescribimos el algoritmo completo:

$\{Q \equiv (0<=i<=N) \wedge (r = \sum \alpha \in \{i+1..N\}.\ v[\alpha])\}$

```
fun iisuma(v : vect[1..n] de ent; i : nat, r : ent) dev (s : ent)
    caso i = 0  →  r
     •    i > 0  →  iisuma(v, i - 1, v[i] + r)
ffun
```

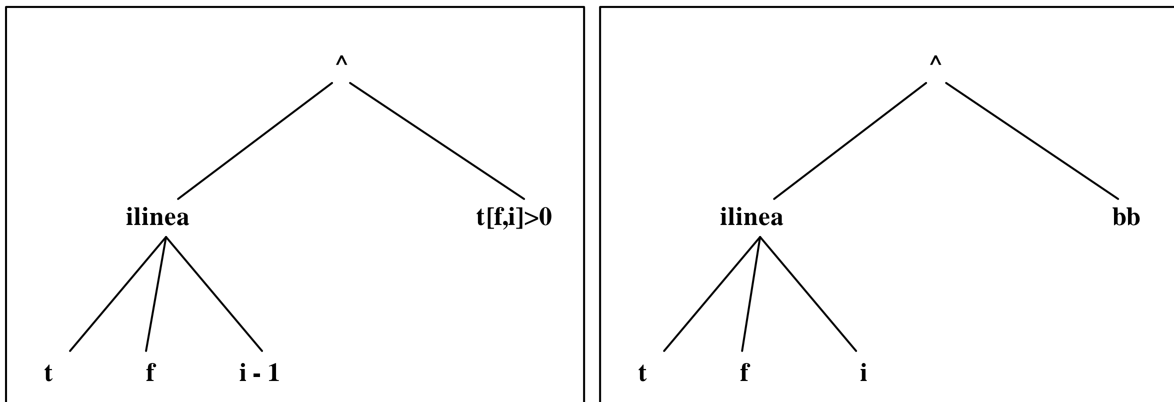
$\{R \equiv s = \sum \alpha \in \{1..N\}.\ v[\alpha]\}$

y como se puede comprobar, la llamada inicial ***iisuma(v, N, 0) = isuma(v, N) = suma(v)*** cumple la precondición ya que para *i=N* y *r=0* tendremos:

$Q \equiv (0<N=N) \wedge (0 = \sum \alpha \in \{N+1..N\}.\ v[\alpha]) \equiv cierto$

<u>Ejemplo2:</u> Apliquemos ahora la técnica de desplegado-plegado al ejemplo2

<u>Generalización:</u>



iilinea(t, f, i, bb) = ilinea(t, f, i) ^ bb

La operación **^** tiene como elemento neutro ***"cierto"*** y es *asociativa.*

Desplegado:

1.- $$\text{iilinea}(t,\ f, i, bb) = \left\{ \begin{array}{l} \text{caso } i = 0 \ \rightarrow \ \text{cierto} \\ \bullet \quad i > 0 \ \rightarrow \text{ilinea}(t,\ f,\ i-1) \wedge \ t[f, i] > 0 \end{array} \right\} \wedge \ bb$$

2.- $$\text{iilinea}(t,\ f, i, bb) = \left\{ \begin{array}{l} \text{caso } i = 0 \ \rightarrow \ \text{cierto} \wedge bb \\ \bullet \quad i > 0 \ \rightarrow \text{ilinea}(t,\ f,\ i-1) \wedge \ t[f, i] > 0 \ \wedge \ bb \end{array} \right\}$$

3.- $$\text{iilinea}(t,\ f, i, bb) = \left\{ \begin{array}{l} \text{caso } i = 0 \ \rightarrow \ bb \\ \bullet \quad i > 0 \ \rightarrow \text{ilinea}(t,\ f,\ i-1) \wedge \ (t[f, i] > 0 \ \wedge \ bb) \end{array} \right\}$$

Plegado:

$$\text{iilinea}(t,\ f, i,\ bb) = \left\{ \begin{array}{l} \text{caso } i = 0 \ \rightarrow \ bb \\ \bullet \quad i > 0 \ \rightarrow \text{iilinea}(t,\ f,\ i-1, t[f, i] > 0 \wedge bb) \end{array} \right\}$$

Como el elemento neutro de la operación "^" es ***cierto***; iilinea(t,f,i,cierto) = ilinea(t,f,i)
y, la ***llamada inicial*** será: ***iilinea(t, f, m, cierto) = ilinea(t, f, m) = linea(t, f)***

Las condiciones adecuadas para bb se deducen de *iilinea(t, f, i, bb) = ilinea(t, f, i) ^ bb* obteniéndose la siguiente:

$$bb = \forall \alpha \in \{i+1..m\}.\ t[f,\alpha] > 0$$ (parámetro acumulador)

Por tanto, la precondición será:

$$\{Q \equiv (1 <= f <= n) \wedge (0 <= i <= m) \wedge (bb = \forall \alpha \in \{i+1..m\}.\ t[f,\alpha] > 0)\}$$

y el código para la función recursiva final obtenida por desplegado-pleagado:

$$\{Q \equiv (1 <= f <= n) \wedge (0 <= i <= m) \wedge (bb = \forall \alpha \in \{i+1..m\}.\ t[f,\alpha] > 0)\}$$

```
fun iilinea(t : vect[1..n, 1..m] de nat; i : nat; bb : bool)  dev  (b : bool)
    caso i = 0  →  bb
     •    i > 0  →  iilinea(t,  f,  i - 1, t[f, i] > 0 ^ bb)
ffun
```

$$\{R \equiv b = \forall \alpha \in \{1..m\}.\ t[f,\alpha] > 0\}$$

EJEMPLO3: SOLUCIÓN PROPUESTA

El algoritmo de la función recursiva final obtenido por desplegado-plegado de la función *iprodesc* será:

$\{Q \equiv (1 <= j <= n) \wedge (pp = \Sigma \alpha \in \{1..j-1\}.\ a[\alpha] * b[\alpha])\}$

fun iiprodesc(a,b: vector[1..n] de ent; j, pp: ent) dev (p: ent)

caso $\mathbf{j = n \rightarrow pp}$

• $\mathbf{j <= n \rightarrow iiprodesc(a, b,\ j+1, pp + (a[j] * b[j]))}$

fcaso

$\{R \equiv p = \Sigma \alpha \in \{1..n\}.\ a[\alpha] * b[\alpha]\}$

y la llamada inicial: ***iiprodesc(a, b, 1, 0) = iprodesc(a, b, 1) = prodesc(a, b)***

CORRECCION DE PROGRAMAS RECURSIVOS

Una vez obtenido el código de la función recursiva, hemos de verificar formalmente su corrección.

Sea la función recursiva lineal:

```
{Q(x̄)}
fun f(x̄ : T1) dev (ȳ :T2)
        caso   Bt(x̄)   →  triv(x̄)
         []    Bnt(x̄)  →  c(f(s(x̄)), x̄)
        fcaso
ffun
{R(x̄, ȳ)
```

los puntos a demostrar para verificar la corrección de la función ***f*** son:

1.- **Completitud de la Alternativa**: $\mathbf{Q(\bar{x}) \Rightarrow B_t(\bar{x}) \vee B_{nt}(\bar{x})}$

Se debe demostrar en este punto que entre los casos triviales y no triviales se cubren todos los casos posibles (todos los valores admitidos como parámetros en la precondición).

2.- **Satisfacción de la Precondición par la llamada interna**: $\mathbf{Q(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow Q(s(\bar{x}))}$

Se ha de demostrar que las llamadas recursivas se realizan con parámetros que cumplen la precondición

3.- **Base de la inducción**: $\mathbf{Q(\bar{x}) \wedge B_t(\bar{x}) \Rightarrow R(\bar{x}, triv(\bar{x}))}$

Se trata de comprobar que para el caso trivial, y con parámetros que cumplan la precondición, la solución que se devuelve cumple la postcondición.
En definitiva: "las soluciones para los casos triviales han de cumplir la postcondición"

4.- **Paso de inducción**: $\mathbf{Q(\bar{x}) \wedge B_{nt}(\bar{x}) \wedge R(s(\bar{x}), \bar{y}') \Rightarrow R(\bar{x}, c(\bar{y}', \bar{x}))}$

Este suele ser el paso mas complicado en el caso de la inmersión no final.

Hay que probar que se cumple la postcondición para los casos recursivos, (no triviales), a partir de la suposición de que la postcondición se satisface para los datos de la llamada interna.
A la expresión $R(s(\bar{x}), \bar{y}')$ se le llama *Hipótesis de Inducción* y se interpreta como que el valor devuelto por la llamada recursiva ($\bar{y}' = f(s(\bar{x}))$) cumple la postcondición *R*
Pues bien, *suponiendo $R(s(\bar{x}), \bar{y}')$ cierto y que se cumple $Q(\bar{x})$ para el caso no trivial $B_{nt}(\bar{x})$, debemos deducir que tras la operación "combinar", se sigue cumpliendo la postcondición R*

5.- **Elección de una estructura de preorden bien fundado (p.b.f.)**:

Hay que encontrar una función *h: D_{T1} → Z* tal que $\mathbf{Q(\bar{x}) \Rightarrow h(\bar{x}) \geq 0}$

Es decir, se trata de encontrar una función de los parámetros de entrada en los enteros no negativos.

Como el *pbf* más conocido son los naturales, en muchas ocasiones es posible dotar a un conjunto de una relación de *pbf* estableciendo una aplicación con el conjunto de los natutares *N*

5.- **Demostración del decrecimiento de los datos:** $\mathbf{Q(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow h(}s\mathbf{(\bar{x})) < h(\bar{x})}$

Se ha de demostrar que en cada llamada recursiva la operación previa que se realiza sobre los parámetros de entrada hace decrecer estrictamente los datos en relación al *pbf* previamente definido.

La función que determina el *pbf* ha de disminuir estrictamente el valor en cada llamada.

Lo que garantizan estos dos últimos puntos (5 y 6) es que la sucesión de llamadas recursivas conducen al caso trivial.

Vamos a comprobar la corrección de las funciones obtenidas en los Ejemplos 1 y 2

Ejemplo1

1.A) Recursiva no final:

```
{Q º0<=i<=N}
  fun isuma(v: vector[1..N] de enteros; i: natural) dev (s: entero)
      caso   i=0    →     0
       []    i>0    →     isuma(v, i-1) + v[i]
      fcaso
  ffun
{R º s = åaÎ {1..i}. v[a]}
```

1.- **Completitud de la Alternativa**: $Q(\vec{x}) \Rightarrow B_t(\vec{x}) \vee B_{nt}(\vec{x})$

(0<=i<=N) Þ (i=0) Ú (i>0) lo que resulta evidente.

"Todos los valores admitidos para i en la precondición se encuentran contemplados, bien en el caso trivial, bien en el no trivial.

2.- **Satisfacción de la Precondición par la llamada interna**: $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \Rightarrow Q(s(\vec{x}))$

En este caso: (0<=i<=N) ∧ (i>0) ⇒ 0<=i-1<=N ; es decir *0<i<=N Þ 0<=i-1<=N*

Al ser *i>0* entonces *i-1>=0* y, por otro lado, como *i<=N* será *i-1<N* por tanto, se puede afirmar que *0<=i-1<=N*

3.- **Base de la inducción**: $Q(\vec{x}) \wedge B_t(\vec{x}) \Rightarrow R(\vec{x}, triv(\vec{x}))$

(0<=i<=N) ***Ù****(i=0)* ***Þ*** *R(v, 0)*

En palabras, se trata de comprobar que la solución para el caso trivial cumple la postondición, es decir que cuando *i=0*, entonces, el valor *s=0* devuelto por la función, cumple la postondición *s=* ***åaÎ*** *{1..0}. v[****a****] = 0*, lo cual resulta evidente al anularse el rango del sumatorio *(convenio sobre cuantificadores).*

4.- **Paso de inducción**: $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \wedge R(s(\vec{x}), \vec{y}') \Rightarrow R(\vec{x}, c(\vec{y}', \vec{x}))$

En este caso: *Q(x)* **º** *(0<=i<=N)* *(1)*
$B_{nt}(x)$ **º** *(i>0)* *(2)*
R(s(x),y') **º** *s' =* ***åaÎ*** *{1..i-1}. v[****a****]* *(3)*
c(y', x) **º** *s' + v[i]*
R **º** *s =* ***åaÎ*** *{1..i}. v[****a****]*

tenemos que comprobar que, cumpliéndose *(1), (2) y(3)*; y teniendo en cuenta los valores de *c(y', x)* y *R*, resulta ser ***s = s' + v[i]***

Como $\boldsymbol{s} = \sum\alpha\in\{1..i\}.v[\alpha] = v[1]+\ldots+v[i-1]+v[i] = (\sum\alpha\in\{1..i-1\}.v[\alpha]) + v[i] = \boldsymbol{s' + v[i]}$

5.- **Elección de una estructura de preorden bien fundado (p.b.f.)**: $Q(\vec{x}) \Rightarrow h(\vec{x}) \geq 0$

Si elegimos *h(v, i) = i* está claro que como *0<=i<=N*, para cualquier valor de *i* de los admitidos por la precondición será *h(v, i) >= 0*

5.- **Demostración del decrecimiento de los datos:** $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \Rightarrow h(s(\vec{x})) < h(\vec{x})$

En este caso: *(0<=i<=N)* ***Ù****(i>0)* ***Þ****h(v, i-1) < h(v, i).*

Como *h(v, i-1) = i-1* y *h(v, i) = i*; para *i>0* resulta evidente que *i-1<i*

1.B) Recursiva final: *(vamos a comprobar la obtenida por plegado-desplegado)*

{Q **º***(0<=i<=N)* ^ *(r =* ***åaÎ*** *{i+1..N}. v[****a****])}*

```
fun iisuma(v : vect[1..n] de ent; i : nat, r : ent) dev (s : ent)
    caso i = 0  →  r
     •   i > 0  →  iisuma(v, i -1, v[i] + r)
ffun
```

{R **º** *s =* ***åaÎ*** *{1..N}. v[****a****]}*

1.- **Completitud de la Alternativa**: $Q(\vec{x}) \Rightarrow B_t(\vec{x}) \vee B_{nt}(\vec{x})$

(0<=i<=N) $\Rightarrow$ *(i=0)* $\vee$ *(i>0)* lo que resulta evidente.

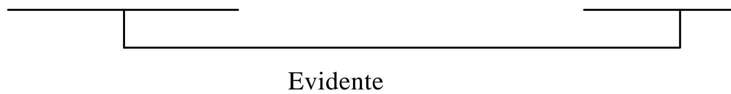
2.- **Satisfacción de la Precondición en la llamada interna**: $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \Rightarrow Q(s(\vec{x}))$
Este es el paso más importante a verificar en el caso de la solución recursiva final ya que se debe garantizar que las llamadas recursivas cumplen la precondición.

En este caso:

(0<=i<=N) ^ (r = ∑α∈ {i+1..N}.v[α]) ^ (i>0) ⇒ (0<=i-1<=N) ^ (r + v[i]= ∑α∈ {i..N}.v[α])

$Q(\overline{x})$ (llamada iisuma(v, i, r)) — $Q(s(\overline{x}))$ (llamada iisuma(v, i-1, r+v[i]))

[(0<=i<=N) ^ (i>0)] ^ (r = ∑α∈ {i+1..N}.v[α])⇒ (0<=i-1<=N) ^ (r + v[i]= ∑α∈ {i..N}.v[α])



Evidente

De *r = v[i+1] + ... + v[N]* , sumando *v[i]* a las dos partes de la igualdad se tiene:

r + v[i] = v[i] + v[i+1] + … + v[N] = *∑α∈{i..N}.v[α])* ; que es lo que se quería demostrar.

3.- **Base de la inducción**: $Q(\vec{x}) \wedge B_t(\vec{x}) \Rightarrow R(\vec{x}, triv(\vec{x}))$

(0<=i<=N) ^ (r = ∑α∈{i+1..N}.v[α]) ^ (i=0) ⇒ *s = ∑α∈{1..N}.v[α]) = r*

Para el caso trivial (*i=0*) tendremos *r = ∑α∈{1..N}.v[α] = s* ; luego bastará que la función *iisuma* devuelva el valor "*r*" como solución que evidentemente cumple la postcondición.

4.- **Paso de inducción**: $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \wedge R(s(\vec{x}), \vec{y}') \Rightarrow R(\vec{x}, c(\vec{y}', \vec{x}))$
En el caso de la recursividad final prácticamente no hay nada que demostrar en este punto ya que si la llamada interna devuelve el valor *y'* que cumple la postcondición *R*, y puesto que no existe función "*combinar*", el valor devuelto cumple la postcondición.

5.- **Elección de una estructura de preorden bien fundado (p.b.f.)**: $Q(\vec{x}) \Rightarrow h(\vec{x}) \geq 0$

Si elegimos *h(v, i) = i* está claro que como *0<=i<=N*, para cualquier valor de *i* de los admitidos por la precondición será *h(v, i) >= 0*

5.- **Demostración del decrecimiento de los datos:** $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \Rightarrow h(s(\vec{x})) < h(\vec{x})$

En este caso: *(0<=i<=N)* ∧ *(i>0)* ⇒ *h(v, i-1) < h(v, i).*

Como *h(v, i-1) = i-1* y *h(v, i) = i*; para *i>0* resulta evidente que *i-1<i*

Ejemplo2

1.A) Recursiva no final:

{Q º(1<=f<=n) ^ (0<=i<=m)}
fun ilinea(t: vector[1..n, 1..m] de nat; f,i: nat) dev (b: bool)
caso i=0 → cierto
[] i>0 → ilinea(t, f, i-1) ^ (t[f,i]>0)
fcaso
ffun
{R º b = "a Î {1..i}. t[f,a]>0}

1.- **Completitud de la Alternativa**: $Q(\vec{x}) \Rightarrow B_t(\vec{x}) \vee B_{nt}(\vec{x})$

(1<=f<=n) ^ (0<=i<=m) Þ (i=0) Ú (i>0) lo que resulta evidente.

2.- **Satisfacción de la Precondición par la llamada interna**: $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \Rightarrow Q(s(\vec{x}))$

En este caso: (1<=f<=n) ^ (0<=i<=m) ∧ (i>0) ⇒ (1<=f<=n) ^ (0<=i-1<=m) ; es decir *(0<i<=m) Þ (0<=i-1<=m)*

Al ser *i>0* entonces *i-1>=0* y, por otro lado, como *i<=m* será *i-1<m* por tanto, se puede afirmar que *0<=i-1<=m*

3.- **Base de la inducción**: $Q(\vec{x}) \wedge B_t(\vec{x}) \Rightarrow R(\vec{x}, triv(\vec{x}))$

(1<=f<=n) ^ (0<=i<=m) ∧ (i=0) ⇒ R(t, f, 0)

En palabras, se trata de comprobar que la solución para el caso trivial cumple la postondición, es decir que cuando i=0, entonces, el valor "cierto" devuelto por la función, cumple la postondición b = ∀α ∈ {1..0}. t[f,α]>0 = cierto, lo cual resulta evidente al anularse el rango del cuantificador universal.

4.- **Paso de inducción**: $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \wedge R(s(\vec{x}), \vec{y}') \Rightarrow R(\vec{x}, c(\vec{y}', \vec{x}))$

En este caso: $Q(\bar{x}) \equiv$ (1<=f<=n) ^ (0<=i<=m) (1)
$B_{nt}(\bar{x}) \equiv$ (i>0) (2)
$R(s(\bar{x}), \bar{y}') \equiv$ b' = ∀α ∈ {1..i-1}. t[f,α]>0 (3)
$c(\bar{y}', \bar{x}) \equiv$ b' ^ t[f,i]>0
R ≡ b = ∀α ∈ {1..i}. t[f,α]>0

tenemos que comprobar que, cumpliéndose *(1), (2) y(3)*; y teniendo en cuenta los valores de *c(y', x)* y *R*, resulta ser ***b = b' ^ t[f, i]>0***

Como R ≡ ***b*** = (∀ α ∈ {1..i}. t[f,α]>0) = (t[f, 1]>0) ^…^ (t[f, i-1]>0) ^ (t[f,i]>0) = (∀α∈{1..i-1}.t[f,α]>0 ^ (t[f,i]>0) = ***b' + t[f,i]>0***

5.- **Elección de una estructura de preorden bien fundado (p.b.f.)**: $Q(\vec{x}) \Rightarrow h(\vec{x}) \geq 0$

Si elegimos h(t, f, i) = i está claro que como 0<=i<=m por precondición, para cualquier valor de i de los admitidos será h(t, f, i) >= 0

6.- **Demostración del decrecimiento de los datos:** $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \Rightarrow h(s(\vec{x})) < h(\vec{x})$

En este caso: (0<=i<=m) ∧ (i>0) ⇒ h(t, f, i-1) < h(t, f, i).

Como h(t, f, i-1) = i-1 y h(t, f, i) = i; para i>0 resulta evidente que i-1<i

2.B) Recursiva final: *(vamos a comprobar la obtenida directamente)*

```
{Q º(1<=f<=n) ^ (0<=i<=m) ^ (bb = "aÎ {1..i}. t[f,a]>0)}
fun ilinea2(t: vector[1..n, 1..m] de nat;f,i: nat; bb: bool) dev (b: bool)
        caso    i=m  →  bb
         []     i<m  →  ilinea2(t, f, i+1, bb ^ t[f,i+1]>0)
        fcaso
ffun
{R º b = "aÎ {1..m}. t[f,a]>0}
```

1.- **Completitud de la Alternativa**: $Q(\vec{x}) \Rightarrow B_t(\vec{x}) \vee B_{nt}(\vec{x})$

En este caso planteamos: <u>*(0<=i<=m) Þ (i=m) Ú (i<m)*</u> lo que resulta evidente.

2.- **Satisfacción de la Precondición en la llamada interna**: $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \Rightarrow Q(s(\vec{x}))$

En este caso: $\underbrace{(1 \leq f \leq n) \wedge (0 \leq i \leq m) \wedge (bb = \forall\alpha \in \{1..i\}.t[f,\alpha] > 0)}_{Q(\bar{x})}$ ^ (i < m)

(llamada ilinea2(t, f, i, bb))

Þ $\underbrace{(1 \leq f \leq n) \wedge (0 \leq i+1 \leq m) \wedge [(bb \wedge t[f, i+1] > 0) = \forall\alpha\in\{1..i+1\}.t[f, \alpha]>0]}_{Q(s(\bar{x}))}$

(llamada iisuma(v, i-1, r+v[i]))

a) [(0≤ i ≤m) ^ (i<m)] ⇒ (0≤ i+1 ≤ m) resulta evidente

b) (bb ∧ t[f, i+1] > 0) ⇒ (∀α∈{1..i}.t[f, α]>0) ^ t[f, i+1] > 0 ⇒ ∀α∈{1..i+1}.t[f, α]>0

que es lo que se quería demostrar.

3.- **Base de la inducción**: $Q(\vec{x}) \wedge B_t(\vec{x}) \Rightarrow R(\vec{x}, triv(\vec{x}))$

(1 ≤ f ≤ n)∧(0 ≤ i ≤ m)∧(bb=∀α∈{1..i}.t[f,α]>0)^(**i=m**) Þ b=∀α∈{1..m}.t[f,α]>0 = bb

Para el caso trivial (i=m) tendremos bb = ∀α∈ {1..m}.t[f,α]>0 por precondición; pero si observamos la postcondición, bastará que la función ilinea2 devuelva el valor "bb" como solución, (se han comprobado todos los valores de la fila f completa), que evidentemente cumple la postcondición.

4.- **Paso de inducción**: $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \wedge R(s(\vec{x}), \vec{y}') \Rightarrow R(\vec{x}, c(\vec{y}', \vec{x}))$

Si por hipótesis de inducción se cumple $R(s(\bar{x}), \bar{y}')$, al no existir operación adicional, el resultado cumple la postcondición

5.- **Elección de una estructura de preorden bien fundado (p.b.f.)**: $Q(\vec{x}) \Rightarrow h(\vec{x}) \geq 0$

Si elegimos h(t, f, i, bb) = m - i está claro que como 0<=i<=m, para cualquier valor de i de los admitidos por la precondición será h(t, f, i, bb) >= 0

6.- **Demostración del decrecimiento de los datos:** $Q(\vec{x}) \wedge B_{nt}(\vec{x}) \Rightarrow h(s(\vec{x})) < h(\vec{x})$

En este caso: (0<=i<=m) ∧ (i<m) ⇒ *h(t, f, i+1, bb ^ t[f, i+1]>0) < h(t, f, i, bb)*.

Como *h(t, f, i+1, bb) = m – (i+1) = (m-i) - 1* y *h(t, f, i, bb) = m - i*;
para 0<=i<m resulta evidente que (m-i) - 1< m - i
y por tanto *h(t, f, i+1, bb ^ t[f, i+1]>0) < h(t, f, i, bb)*

EJEMPLOS Y EJERCICIOS:

De la ***colección de problemas curso 1999-2000*** conviene hacer los ejercicios **2.2.1, 7.3, 8.2, 8.3, 8.4, 9.1, 9.2, 9.3, 9.4, 9.5**

Todo lo que caiga en tus manos sobre ejercicios relacionados con el diseño y verificación de algoritmos recursivos.

DISEÑO ITERATIVO

SEMÁNTICA DEL LENGUAJE

En el lenguaje imperativo dado un *estado inicial*, la ejecución de un programa es determinista en el sentido de que la secuencia de estados por la que pasa, incluido el propio *estado final*, están completamente determinados por el estado de partida.

Al conjunto de estados alcanzados por el algoritmo en un punto intermedio de su ejecución se les llama "*aserciones o asertos*"

Consideramos un programa de la forma $S = s_1;\ldots s_n$ cuya especificación es {Q} S {R}. Al introducir asertos intermedios obtenemos $\{Q \equiv P\} s_1 \{P_1\} s_2 \{P_2\} \ldots \{P_{n-1}\} s_n \{P_n \equiv R\}$

Las reglas del lenguaje imperativo se darán de la forma $\frac{E_1, \ldots, E_n}{E'_1, \ldots, E'_m}$ con n>=0 y m>=1 y se lee: "*De la veracidad de las expresiones* $E_1,\ldots E_n$ *se infiere la veracidad de las expresiones* $E_1,\ldots E_m$

AXIOMAS

1.- **Una precondición siempre se puede reforzar.** $\frac{\{Q\}S\{R\},\ \ Q' \Rightarrow Q}{\{Q\}S\{R\}}$

Si Q define un conjunto de estados admisibles como precondición de un algoritmo, también será admisible cualquier subconjunto suyo

En particular, "*falso*" siempre es una precondición admisible aunque carece de interés ya que, al definir el conjunto vacío de estados, el algoritmo será inútil.

2.- **Una postcondición siempre se puede debilitar.** $\frac{\{Q\}S\{R\},\ \ R \Rightarrow R'}{\{Q\}S\{R'\}}$

Si R define un conjunto de estados admisibles como postcondición de un algoritmo, cualquier "superconjunto" suyo también será admisible

En particular, *cierto* siempre es una postcondición admisible si el programa termina.

3.- **Las precondiciones admisibles para un algoritmo pueden componerse mediante las conectividades Ù y Ú siempre que se haga lo mismo en las respectivas postcondiciones**

$$\frac{\{Q_1\}S\{R_1\},\ \ \{Q_2\}S\{R_2\}}{\{Q_1 \wedge Q_2\}S\{R_1 \wedge R_2\},\ \ \{Q_1 \vee Q_2\}S\{R_1 \vee R_2\}}$$

INSTRUCCIONES

Nada: Los conjuntos inicial y final de estados coinciden, es decir, la instrucción nada siempre termina y su efecto sobre el estado del cómputo es nulo

$$\frac{}{\{Q\}nada\{Q\}} \equiv \frac{Q \Rightarrow R}{\{Q\}nada\{Q\}} \qquad \text{cualquiera que sea Q}$$

Abortar: No existe ningún estado de partida que garantice que la instrucción abortar termina en un estado definido

$$\overline{\{falso\}abortar\{Q\}} \text{ cualquiera que sea Q}$$

Asignación: Para cualquier postcondición R: $\overline{\{Dom(E) \wedge R_x^E\}x := E\{R\}}$

R_x^E se construye sustituyendo en *R* todas las apariciones *libres* de *x* por la expresión *E*

Dom(E) es el conjunto de todos los estados en los que la expresión *E* está definida.

La regla permite calcular una precondición admisible a partir de una postcondición.

Ejemplo: En la especificación {Q} x:=x+1 {R}, calqular Q para distintas postcondiciones R:

a) R≡ x=7

{Q}
x := x+1 Q ≡ R_x^{x+1}≡ x+1=7 ≡ x=6
{R≡ x=7}

b) R≡ x+y>0 Q ≡ R_x^{x+1}≡ x+1+y>0 ≡ x+y>=0

c) R≡ y=2k Q ≡ R_x^{x+1}≡ y=2k (la expresión no contiene x)

d) R≡ (∃x>=0. y=2x) Q ≡ R_x^{x+1}≡ (∃x>=0. y=2x) (x no es libre)

Cuando aparezcan en la expresión *E* componentes de un vector o cualquier tipo de operaciones parciales, se hará constar expresamente en la precondición el dominio de E

Se admiten ***asignaciones múltiples***, esto es, la asignación "*simultánea*" de una lista de expresiones a una lista de variables de los tipos adecuados. Por *ejemplo* <x,y> := <y,x> permuta los viejos valores de x e y evitando sobreespecificar el orden en que han de tener lugar el conjunto de asignaciones. Así si x=6 e y=4 la asignación múltiple provoca que x=4 e y=6 que no es el mismo resultado que x:= y; y:= x secuencialmente ya que esto daría como resultado 4 tanto para x como para y.

En general: $\overline{\{R_{x_1 \ldots x_n}^{E_1 \ldots E_n}\} < x_1, \ldots x_n > := < E_1, \ldots E_n > \{R\}}$

Composición secuencial de instrucciones: $\dfrac{\{Q\}S_1\{P\},\ \{P\}S_2\{R\}}{\{Q\}S_1;S_2\{R\}}$

Si S_1 y S_2 son asignaciones se calculan precondiciones en sentido inverso al del flujo del programa: partiendo de R, se calcula P mediante la regla de asignación y, tomando P como postcondición de S_1, se aplica de nuevo la regla de asignación para calcular Q

Ejemplo: Hallar Q en el algoritmo siguiente:

(Calcular la precondición conocida la postcondición)

{Q}		{Q}
x:= x+y;		x:= x+y;
y:= x-y;	escribimos	{P}
x:= x-y;		y:= x-y;
{R≡ (x=A) ∧ (y=B)		{S}
		x:= x-y;
		{R≡ (x=A) ∧ (y=B)

y calculamos en sentido inverso

$\{S\} \equiv R_{<x,y>}^{<x-y,y>} \equiv (x\text{-}y=A) \wedge (y=B) \equiv (x=A+B) \wedge (y=B)$

$\{P\} \equiv S_{<x,y>}^{<x,x-y>} \equiv (x=A+B) \wedge (x\text{-}y=B) \equiv (x=A+B) \wedge (y=A)$

$\{Q\} \equiv P_{<x,y>}^{<x+y,y>} \equiv (x+y=A+B) \wedge (y=A) \equiv (x=B) \wedge (y=A)$

INSTRUCCIONES CONDICIONALES

A.- **Si B entonces S_1 sino S_2 fsi** $\dfrac{\{Q \wedge B\}S_1\{R\};\ \{Q \wedge \neg B\}S_2\{R\}}{\{Q \wedge Dom(B)\}\ si\ B\ entonces\ S_1\ sino\ S_2 fsi\{R\}}$

La regla obliga a que se demuestre que se satisfacen las dos especificaciones del antecedente de la regla.

Se puede partir también de la postcondición R, calculando hacia atrás las precondiciones. Es decir, partiendo de R se calcula Q_1 para B y Q_2 para ØB. Obtenidas ambas se calcula la precondición de "Si B entonces..." como:

{Dom(B) Ù((Q1 ÙB) Ú(Q2 ÙØB))}

Ejemplo:

{Q}
si x<0 entonces x:=x+1 sino x:=x-1 fsi
{R≡ x>=0}

Para x<0 tenemos {Q_1}
x:= x+1
{R≡ x>=0}

luego hacia atrás calculamos *Q_1* º x+1>=0 ≡ *x>=-1*

Para x>=0 tenemos {Q_2}
x:= x-1
{R≡ x>=0}

luego hacia atrás calculamos Q_2 ≡ x-1>=0 ≡ *x>=1*

entonces: *Q* ≡(x>=-1 ∧ x<0) ∨ (x>=1 ∧ x>=0) ≡ *(x=-1)* ∨*(x>=1)*

B.- La segunda forma de instrucción condicional es la versión imperativa de la expresión "*caso*": "**caso B1→S1 …. casoBn→Sn fcaso**"; (ver en el libro las reglas)

INSTRUCCIONES ITERATIVAS: INVARIANTES

A.- "**mientras B hacer S fmientras**"

Comienza evaluando la condición booleana B:

- *si B es falsa, el bucle equivale a la instrucción nada*
- *en otro caso se ejecuta el cuerpo S de la iteración*

No es práctico dar una regla que permita calcular la precondición a partir de la postcondición.

Invariante: Es un predicado que describe todos los estados por los que atraviesa el cómputo realizado por el bucle, observado *justo antes de evaluar la condición B* de terminación.

El invariante se satisface antes de la primera iteración, después de cada una de ellas y, en particular, después de la última, es decir, a la terminación del bucle.

1.- *Si el bucle termina*, lo hace satisfaciendo el invariante *P* y la negación de la condición *B* Por tanto, ***R* •*P* ∧¬*B***

2.- Además, el invariante *P* es a la vez ***precondición*** y ***postcondición*** del cuerpo *S* de la iteración (se debe cumplir antes de la primera iteración y después de cada una de ellas).

En consecuencia se puede enunciar la siguiente regla: "*Comenzando la ejecución del bucle en estados que cumplen P, si el bucle termina, lo hace cumpliendo P* ∧¬*B* :

$$\textbf{REGLA}: \quad \frac{\{P \wedge B\} S \{P\}}{\{P\}\ mientras\ B\ hacer\ S\ fmientas \{P \wedge \neg B\}}$$

Pero la regla es **incompleta** ya que al exigir tan sólo que *S* mantenga la invarianza de *P* no garantizamos que el bucle termina, sólo afirmamos que *si termina lo hace…*

Para resolver el problema debemos encontrar una función *t: estados* → *Z* (*función limitadora*) que se mantenga *no negativa* mientras se realiza el cuerpo *S* y que *decrezca* cada vez que se ejecuta éste:

1.- $P \wedge B \Rightarrow t>=0$

2.- $\{P \wedge B \wedge t=T\}\ S\ \{t<T\}$ *para cualquier constante entera T*

Así la **regla completa** será: $\dfrac{\{P \wedge B\}S\{P\},\quad P \wedge B \Rightarrow t \geq 0,\ \ \{P \wedge B \wedge t = T\}S\{t < T\}}{\{P\}\ mientras\ B\ hacer\ S\ \ fmientas\{P \wedge \neg B\}}$

Para encontrar funciones limitadoras:

- Observar las *variables que son modificadas* por el cuerpo *S* del bucle
- Construir con (algunas de) ellas una expresión entera *t* que decrezca
- Ajustar *t*, si fuera necesario, para garantizar que se mantiene *no negativa* siempre que se cumple $P \wedge B$

B.- "**para i desde E_1 hasta E_2 hacer**
S
fpara" equivale a
i:= E_1 ; lim:= E_2;
mientras i $\leq$ lim hacer
S; i:=i+1;
fmientras

o

"**para i bajando desde E_1 hasta E_2 hacer**
S
fpara" equivale a
i:= E_1 ; lim:= E_2;
mientras i $\geq$ lim hacer
S; i:=i-1;
fmientras

donde *lim* es un nombre de variable que no ha de coincidir con ningún otro nombre de variables consultadas o modificadas por *S*

En lugar de reglas específicas para la instrucción para , se utilizan las correspondientes equivalencias.

ESQUEMA DE PROGRAMA ITERATIVO

{Q}
Inic;
mientras B hacer {P}
S
fmientras
{R}

"El invariante *P* ha de ser lo *suficientemente fuerte* para que, junto con $\neg B$ conduzca a la postcondición *R*, y lo *suficientemente débil* para que las instrucciones *Inic* de inicialización, hagan que se satisfaga antes de la primera iteración, y, además, el cuerpo del bucle lo mantenga invariante".

Una técnica para "*adivinar*" invariantes consiste en tabular, para unas cuantas iteraciones, los valores de las variables modificadas por el bucle. De esta forma se clarifican las relaciones entre ellas.
Algunas de estas relaciones formarán parte del invariante. El resto de condiciones (relaciones) para el invariante se irán descubriendo después a medida que se necesiten propiedades para completar las demostraciones de su verificación formal.

VERIFICACIÓN FORMAL

Puntos a demostrar:
0.- Inventar un invariante P y una función limitadora t
1.- $P \wedge \neg B \Rightarrow R$
2.- $\{Q\}\ Inic\ \{P\}$
3.- $\{P \wedge B\}\ S\ \{P\}$
4.- $P \wedge B \Rightarrow t \geq 0$
5.- $\{P \wedge B \wedge t=T\}\ S\ \{t<T\}$

que son un resumen de lo que hemos visto hasta el momento.

Si el algoritmo es complejo puede descomponerse en pequeños algoritmos, teniendo en cuenta:

Si consta de varios bucles en secuencia, basta con inventar predicados intermedios que sirvan de postcondición de un bucle y, a la vez, como precondición del siguiente. Después se verificará cada bucle por separado.

Si consta de varios bucle anidados, se comienza por verificar el más interior. A efectos de verificar el bucle inmediatamente más externo, el bucle interno se trata como una instrucción simple.

EJEMPLOS Y EJERCICIOS:

De la ***colección de problemas curso 1999-2000*** conviene hacer los ejercicios **3.1, 7.1, 7.2, 7.4, 8.1, 8.5, 8.6**

Del libro de texto ejercicios **4.2 y 4.4**

DERIVACIÓN FORMAL DE PROGRAMAS ITERATIVOS

La técnica cubre dos posibilidades:

a) DERIVACIÓN DE INSTRUCCIONES SIMPLES

Se trata básicamente de ir razonando hacia atrás, partiendo de la postcondición y comprobando si la precondición *Q* de la instrucción garantiza *Dom(E)* **Ù** R_x^E

Ejemplo: Consideremos la siguiente especificación:

> $\{Q \equiv a=A \wedge b=B \wedge AB=u+ab\}$
> *instrucción a derivar*
> $\{R \equiv a=Adiv2 \wedge b=2B \wedge AB=u+ab\}$

Ensayamos <a,b>:= <Adiv2, 2B> y obtenemos:
$R_{a,b}^{Adiv2,2B} \equiv (Adiv2=Adiv2) \wedge (2B=2B) \wedge (AB = u + (Adiv2)2B) \equiv AB = u + (Adiv2)2B$

¿Es cierto que $Q \Rightarrow Dom(E) \wedge R_x^E$?
$(a=A) \wedge (b=B) \wedge (AB=u+ab) \Rightarrow AB = u + (Adiv2)2B$, es decir u+AB = u+ (Adiv2)2B
Esto será cierto sólo si A es par pero no si es impar. Si A es impar (Adiv2)*2 = A-1 y por tanto u+ (Adiv2)2B = u+(A-1)B = u+AB-B

La instrucción resultante debe ser:

> *caso par(n) → <a, b>:= <Adiv2, 2B>*
> *caso impar(n) → <a, b, u>:= <Adiv2, 2B, u+B>*
> *fcaso*

B.- DERIVACIÓN DE BUCLES

Si hubiera necesidad de bucles anidados, se deriva cada uno por separado y se procede en sentido descendiente, es decir, derivando primero el más externo y progresivamente los internos.

<u>*B1.- Derivación de bucles a partir de invariantes:*</u>

1.- Conociendo P y R determinar cuál ha de ser ¬B para que *P* **ÙØ***B* **Þ** *R*

   A partir de ¬B se obtiene B

2.- Conociendo P se determinamos las instrucciones de *Inicialización* que hacen *{Q} Inic {P}*

   Estas instrucciones serán lo más simple posibles.

   Es posible que tengamos que hacer algunos ajustes a la precondición Q

3.- Hasta el momento, hemos derivado

{Q}
Inic;
mientras B hacer {P}
¿?
fmientras
{R}

- Para derivar el cuerpo del bucle S:

a) Incluimos al final de S, una instrucción "*Avanzar*" que haga progresar el bucle hacia su terminación. *La instrucción Avanzar ha de hacer decrecer la función limitadora t*

b) Normalmente, la instrucción <u>Avanzar rompe la invarianza de P</u> por lo que es necesario incluir otra instrucción "*Restablecer*" para que se siga cumpliendo *{P Ù B} S {P}*

Si comprobamos que {P ∧ B} ⇒ T entonces Restablecer sería "nada"

<u>B.2.- Derivación del invariante a partir de la postcondición</u>

- La idea de partida es que el invariante es un predicado más débil que la postcondición. *Sólo algunos estados de P son también estados de R* (los que cumplen además ¬B)

- Como conocemos *R*, y deseamos conocer *P*, hemos de proceder a <u>debilitar *R*</u>. ¿Hasta dónde?. *Hasta que R debilitada sea P*, es decir, hasta que *P* sea tan débil para que alguno de sus estados pueda establecerse fácilmente con unas simples asignaciones

- Las técnicas para debilitar un predicado son las que ya vimos al estudiar las técnicas de inmersión:

  a) *<u>Elminar una conjunción</u>*: Si *R* es de la forma *R1 Ù R2*, una de las dos conjunciones (la que más fácil sea de establecer al comienzo del bucle) se tomará como *P*. La otra será directamente *ØB* y de esta forma *P Ù ØB* conducirá obviamente a *R*

  b) *<u>Incorporar una variable</u>*: Sustituir en *R* cierta expresión por una nueva variable. Así *R* se escribirá como una conjunción.

- Normalmente, el invariante delimita los valores permitidos para todas las variables que intervienen en el bucle.

<u>Ejemplo1</u>

Es hora de retomar el ejemplo del algoritmo que calculaba la suma de los elementos de un vector, cuya especificación era:

{Q º cierto}
fun suma(v: vector[1..n] de ent) dev (s: ent)
{R º s = åa Î {1..N}. v[a]}

Vamos a derivar un programa iterativo que resuelva el problema.

- Iniciamos el proceso *debilitando la postcondición*. Para ello introducimos una variable que sustituya la expresión constante *N*.

  Tendremos: $R \equiv (s = \sum \alpha \in \{1..i\}.\ v[\alpha]) \wedge (i = N)$. De esta forma $R \equiv R_1 \wedge R_2$.

  Si tomamos $\{P \equiv (s = \sum \alpha \in \{1..i\}.\ v[\alpha])\}$ y $\neg B \equiv (i = N)$ es obvio que $P \wedge \neg B \Rightarrow R$

  Lógicamente $B \equiv i \neq N$

- Hasta ahora tenemos:

```
{Q ≡ cierto}
fun sumait(v: vector[1..n] de ent) dev (s: ent)
        Inicializar;
        mientras i ≠ N hacer    {P ≡ (s = Σα∈{1..i}. v[α])}
                Restablecer;
                Avanzar;
        fmientras
ffun
{R ≡ s = Σα∈{1..N}. v[α]}
```

- Tenemos que diseñar los términos *Inicializar*, *Avanzar* y *Restablecer*

  Inicializar supone declarar las dos variables implicadas en el invariante: "*s*" e "*i*" asignándole valores que hagan que antes de iniciarse el bucle se cumpla *P*. Esto último se puede lograr fácilmente en nuestro caso si damos el valor **0** a la variable ***i*** ya que sabemos que en ese caso ***s*** también valdrá **0**.

  Por tanto *Inicializar* ≡ *var i: nat, s: ent fvar*
  <s, i>:= <0, 0>

  El término *Avanzar* ha de modificar las variables (o alguna de ellas) para que se acerque el bucle a su condición de finalización.

  En este caso ***i*** parte del valor **0** y la condición de finalización es ***i = N***. Parece lógico que hagamos: $Avanzar \equiv i := i+1$

  El paso "más complicado" es establecer el término *Restablecer*. Como queremos que $\{P \wedge B\}\ S\ \{P\}$, planteamos $\{P \wedge B\} \Rightarrow P_{s,i}^{s',i+1}$

  por un lado $\{P \wedge B\}$ será $(s = \sum \alpha \in \{1..i\}.\ v[\alpha]) \wedge (i \neq N)$ y por otro

  $$P_{s,i}^{s',i+1} \equiv s' = \sum \alpha \in \{1..i+1\}.\ v[\alpha] = (\sum \alpha \in \{1..i\}.\ v[\alpha]) + v[i+1] = s + v[i+1]$$

  Se deduce por tanto que la instrucción *Restablecer* ha de consistir en sumar a ***s*** el término *v[i+1]*, luego tendremos: $Restablecer \equiv s := s+v[i+1]$

Con esto habremos terminado el diseño formal y tendremos:

$\{Q \equiv cierto\}$
fun sumait(v: vector[1..n] de ent) dev (s: ent)
  var i: nat, s: ent fvar
  <s, i>:= <0, 0>
  mientras $i \neq N$ *hacer* $\{P \equiv (s = \Sigma\alpha \in \{1..i\}.\ v[\alpha])\}$
    s:= s+v[i+1];
    i:= i+1;
  fmientras
  dev s
ffun
$\{R \equiv s = \Sigma\alpha \in \{1..N\}.\ v[\alpha]\}$

VERIFICACIÓN FORMAL:

0.- Fijar invariante y función limitadora:

El invariante se ha obtenido en la fase de diseño luego no tenemos que hacer nada .

La función limitadora puede ser $t(v, s, i) = N-i$ que cumple $t \geq 0 \quad (0 \leq i \leq N)$

1.- $P \wedge \neg B \Rightarrow R$ Es cierto por construcción

2.- {Q} Inic {P} $0 = \Sigma\alpha \in \{1..0\}.\ v[\alpha]$ Evidente.

3.- $\{P \wedge B\}$ S {P} "*Teniendo en cuenta que a la entrada del bucle se cumple el invariante (P) y la condición de bucle (B), tras las operaciones interiores al bucle se ha de cumplir el invariante para los nuevos valores de las variables*"

Precisamente para calcular el término *Restablecer* hemos utilizado esta propiedad. Por tanto cuando empleemos este método de derivación la propiedad será cierta y no tendremos que demostrar nada aquí.

4.- $P \wedge B \Rightarrow t \geq 0$ *"Verficar que la función limitadora se mantiene no negativa durante la ejecución del bucle".*

Cuestión ya comentada en el punto 0.

5.- $\{P \wedge B \wedge t=T\}$ S $\{t<T\}$ "*Verificar que, a cada paso del bucle, la función limitadora decrece estrictamente"*

Evidentemente $n - (i+1) < n-i$

Ejemplo 2

ESPECIFICACIÓN: $\{Q \equiv 1<=f<=n\}$
fun linea(t: vector[1..n, 1..m] de nat; f: nat) dev (b: bool)
$\{R \equiv b = \forall \alpha \in \{1..m\}.\ t[f,\alpha]>0\}$

- Partiendo de la postcondición, la debilitamos introduciendo un parámetro variable ***i*** en sustitución de la constante ***m***

$$R \equiv R_1 \wedge R_2 \quad \text{CON} \quad R_1 \equiv (b = \forall \alpha \in \{1..i\}.\ t[f,\alpha]>0) \quad y \quad R_2 \equiv (i = m)$$

Tomamos la primera como invariante del bucle y la segunda como ¬B y tendremos

```
{Q ≡ 1<=f<=n}
fun lineait(t: vector[1..n, 1..m] de nat; f: nat) dev (b: bool)
Inicializar;
mientras i ≠ m hacer  {P ≡ (b = ∀α ∈ {1..i}. t[f,α]>0)}
        Restablecer;
        Avanzar;
fmientras
ffun
{R ≡ b = ∀α ∈ {1..m}. t[f,α]>0}
```

- *Inicializar*: Cuando $i=0$ sabemos que $b = \forall \alpha \in \{1..0\}.\ t[f,\alpha]>0$ es "*cierto*"; luego

$$Inicializar \equiv \langle i, b\rangle := \langle 0, cierto\rangle$$

- *Avanzar*: Si *i* comienza tomando el valor *0* y debe alcanzar el valor *n*, el termino *Avanzar* podría ser:

$$Avanzar \equiv i := i+1$$

- *Restablecer*: Tendrá que cumplirse $\{P \wedge B\}\ S\ \{P\}$, planteamos $\{P \wedge B\} \Rightarrow P_{b,i}^{b',i+1}$

Por un lado $\{P \wedge B\} \equiv (b = \forall \alpha \in \{1..i\}.\ t[f,\alpha]>0) \wedge (i < m)$

(i≠m lo hemos cambiado por i<m ya que en este caso es equivalente)

Por otro

$$P_{b,i}^{b',i+1} \equiv b' = \forall \alpha \in \{1..i+1\}.\ t[f,\alpha]>0 \equiv b'=(\forall \alpha \in \{1..i\}.\ t[f,\alpha]>0) \wedge (t[f, i+1]>0) \equiv b' = b \wedge (t[f, i+1]>0)$$

Si hacemos $b' := b \wedge (t[f, i+1]>0)$ se tiene $\{P \wedge B\} \Rightarrow P_{b,i}^{b',i+1}$; por tanto

$$Restablecer \equiv b := b \wedge (t[f, i+1]>0)$$

- El algoritmo iterativo:

```
{Q ≡ 1<=f<=n}
fun lineait(t: vector[1..n, 1..m] de nat; f: nat) dev (b: bool)
var  i: nat;  b: bool  fvar;
<i, b>:= <0, cierto>
```

```
mientras  i < m  hacer   {P º(b =  "a Î {1..i}. t[f,a]>0)}
          b:= b Ù(t[f, i+1]>0);
          i:= i+1;
fmientras;
dev b
ffun
{R º b =  "a Î {1..m}. t[f,a]>0}
```

Esta sería una solución válida, no obstante vamos a plantearnos un par de mejoras respecto al algoritmo obtenido.

1.- Si en la especificación de la función:

```
{Q º1<=f<=n}
fun linea(t: vector[1..n, 1..m] de nat; f: nat) dev (b: bool)
{R º b =  "a Î {1..m}. t[f,a]>0}
```

introducimos el predicado auxiliar *lcompleta(t, f, i) = ("a Î {1..i}. t[f,a]>0)*, el diseño iterativo que tendríamos ahora sería:

```
P º b = lcompleta(t, f, i)
B º i < m
R º b = lcompleta(t, f, m)
```

En principio el rango de ***i*** sería *1 £ i £ m*, pero se puede ampliar con *i = 0* ya que para dicho valor sabemos que *lcompleta(t, i, 0) = cierto.*

En este caso incluiremos en *P* el rango de *i* con lo que tendremos

P º b = lcompleta(t, f, i) Ù(0 £ i £ m).

Los términos *Inicializar*, *Avanzar* y *Restablecer* no sufrirán variación por lo que el algoritmo quedará:

```
{Q º1<=f<=n}
fun lineait2(t: vector[1..n, 1..m] de nat; f: nat) dev (b: bool)
var   i: nat;   b: bool   fvar;
<i, b>:= <0, cierto>
mientras  i < m  hacer   { P º b = lcompleta(t, f, i) Ù(0 £ i £ m)}
          b:= b Ù(t[f, i+1]>0);
          i:= i+1;
fmientras;
dev b
ffun
{R º b = lcompleta(t, f, m)}
```

2.- Por otra parte, en el momento en que se detecta que un elemento *t[f, a]* es *cero* se puede afirmar que *b* será *falso* y no se debería seguir iterando.

Esto equivale a poner una condición de finalización de bucle que compruebe a cada paso del bucle si *b* sigue siendo cierto.

La función modificada sería:

```
{Q ≡ 1<=f<=n}
fun lineait3(t: vector[1..n, 1..m] de nat; f: nat) dev (b: bool)
var   i: nat;   b: bool   fvar;
<i, b>:= <0, cierto>
mientras  ((i < m) ∧ b)  hacer   { P ≡ b = lcompleta(t, f, i) ∧ (0 ≤ i ≤ m)}
        b:= b ∧ (t[f, i+1]>0);
        i:= i+1;
fmientras;
dev b
ffun
{R ≡ b = lcompleta(t, f, m)}
```

VERIFICACIÓN:

1.- $P \wedge \neg B \Rightarrow R$ Tanto para *lineait* como *lineait2*, por construcción, es evidente.

Para *lineait3* tendremos:

$b = lcompleta(t, f, i) \wedge (0 \le i \le m) \wedge ((i = m) \vee \neg b) \Rightarrow b = lcompleta(t, f, m)$

a) $b = lcompleta(t, f, i) \wedge (i = m) \Rightarrow b = lcompleta(t, f, m)$ resulta evidente.

b) $b = lcompleta(t, f, i) \wedge falso \Rightarrow b = lcompleta(t, f, m)$

$falso \quad \Rightarrow \quad ¿?$ *siempre se cumple.*

2.- {Q} Inic {P} *cierto = lcompleta(t, f, 0)* en cualquier caso

3.- $\{P \wedge B\}$ S {P} Se trata del punto más conflictivo en el proceso de verificación

En el caso de *lineait* se tiene:

$\{P \wedge B\} \equiv (b = \forall \alpha \in \{1..i\}.\ t[f,\alpha]>0) \wedge (0 \le i < m)$

$P_{b,i}^{b',i+1} \equiv (b' = \forall \alpha \in \{1..i+1\}.t[f,\alpha]>0) \wedge (0 \le i+1 \le m) \equiv$
$(b' = b \wedge (t[f, i+1>0]) \wedge (0 \le i+1 \le m)$

Tenemos que
$(0 \le i < m) \Rightarrow (0 \le i+1 \le m)$ y $(b = \forall \alpha \in \{1..i\}.t[f,\alpha]>0) \Rightarrow (b' = b \wedge (t[f, i+1>0])$

luego $\{P \wedge B\} \Rightarrow P_{b,i}^{b',i+1}$

Para *lineait2* tenemos:

$\{P \wedge B\} \equiv (b = lcompleta(t, f, i)) \wedge (0 \le i < m)$

$P_{b,i}^{b',i+1} \equiv (b' = lcompleta(t, f, i+1)) \wedge (0 \le i+1 \le m) \equiv (b' = b \wedge t[f, i+1>0]) \wedge (0 \le i+1 \le m)$

razonando como antes $\{P \wedge B\} \Rightarrow P_{b,i}^{b',i+1}$, y esto equivale a $\{P \wedge B\}\ S\ \{P\}$

Finalmente en el caso de *lineait3* tenemos:

$$\{P \wedge B\} \equiv (b=lcompleta(t,f,i)) \wedge (0 \le i \le m) \wedge ((i< m) \wedge b) \equiv (\underline{b=lcompleta(t,f,i)}) \wedge (0 \le i<m) \wedge b$$

$$P^{b',i+1}_{b,i} \equiv (b' = lcompleta(t, f, i+1)) \wedge (0 \le i+1 \le m) \wedge b' \equiv$$
$$(b'= \underline{lcompleta(t, f, i)} \wedge t[f, i+1>0]) \wedge (0 \le i+1 \le m) \wedge b' \equiv$$
$$(\underline{b'= b \wedge t[f, i+1>0]}) \wedge (0 \le i+1 \le m) \wedge (\underline{b \wedge t[f, i+1>0]})$$

a cada paso del bucle, tras las instrucciones *Restablecer* y *Avanzar* $<b;\ i>:=\ <b \wedge t[f,\ i+1]>0;\ i+1>$ se cumple el invariante *P*, luego podemos concluir que $\{P \wedge B\}\ S\ \{P\}$ es cierto

4.- $P \wedge B \Rightarrow t \ge 0$ Si tomamos $l(t, f, i) = m-i$, puesto que el valor inicial de *i* es *0* y a cada paso del bucle se incrementa en *1* hasta alcanzar el valor *m*, se tiene $l(t, f, i) \ge 0$

5.- $\{P \wedge B \wedge t=T\}\ S\ \{t<T\}$ Es evidente que $l(t, f, i) > l(t, f, i+1)$ por ser $m-i-1 < m-i$

EJEMPLOS Y EJERCICIOS:

De la ***colección de problemas curso 1999-2000*** conviene hacer los ejercicios **3.2.1, 11.1, 11.2, 11.3**

Exámenes de años anteriores

TRANSFORMACIONES RECURSIVO-ITERATIVO

El diseño recursivo puede utilizarse como técnica para el descubrimiento de invariantes

Las razones para desear transformar a iterativo una función recursiva son:

- El lenguaje iterativo disponible no soporta la recursividad
- No se quiere pagar el coste adicional en tiempo del mecanismo de llamada a procedimiento y del paso de parámetros
- No se quiere pagar el coste adicional de memoria que lleva implícita la implementación de la recursividad.

Conviene tener en cuenta que la versión iterativa siempre será menos legible y modificable.

A) TRANSFORMACIÓN RECURSIVA FINAL – ITERATIVA

$\{Q(\overline{x})\}$
fun $f(\overline{x}:T_1)$ *dev* $(\overline{y}:T_2) =$
caso $Bt(\overline{x}) \rightarrow triv(\overline{x})$
$Bnt(\overline{x}) \rightarrow f(s(\overline{x}))$
fcaso
ffun
$\{R(\overline{x},\overline{y})\}$

$\{Q(\overline{x_{ini}})\}$
fun $f(\overline{x_{ini}}:T_1)$ *dev* $(\overline{y}:T_2)$
var $\overline{x}:T_1$ *fvar*
$\overline{x} := \overline{x_{ini}};$
$\{P(\overline{x},\overline{x_{ini}})\}$
mientras $Bnt(\overline{x})$ *hacer*
$\overline{x} := s(\overline{x})$
fmientras
dev $triv(\overline{x})$
ffun
$\{R(\overline{x_{ini}},\overline{y})\}$

donde $P(\overline{x},\overline{x_{ini}}) \equiv Q(\overline{x}) \wedge f(\overline{x_{ini}}) = f(\overline{x})$

B) TRANSFORMACIÓN RECURSIVA NO FINAL – ITERATIVA

$\{Q(\overline{x})\}$
fun $f(\overline{x}:T_1)$ *dev* $(\overline{y}:T_2)=$
caso $Bt(\overline{x}) \rightarrow triv(\overline{x})$
$Bnt(\overline{x}) \rightarrow c(f(s(\overline{x})),\overline{x})$
fcaso
ffun
$\{R(\overline{x},\overline{y})\}$

$\{Q(\overline{x_{ini}})\}$
fun $f(\overline{x_{ini}}:T_1)$ *dev* $(\overline{y}:T_2)$
var $\overline{x}:T_1$ *fvar*
$\overline{x} := \overline{x_{ini}};$
$\{P_1(\overline{x},\overline{x_{ini}})\}$
mientras $Bnt(\overline{x})$ *hacer*
$\overline{x} := s(\overline{x})$
fmientras;
$\overline{y} := triv(\overline{x})$
$\{P_2(\overline{x},\overline{x_{ini}},\overline{y})\}$
mientras $\overline{x} \neq \overline{x_{ini}}$ *hacer*
$\overline{x} = s^{-1}(\overline{x});$
$\overline{y} := c(\overline{y},\overline{x})$
fmientras
dev $\overline{y}$
ffun
$\{R(\overline{x_{ini}},\overline{y})\}$

donde s^{-1} *es la inversa de la función sucesor*

$P_1(\overline{x},\overline{x_{ini}}) \equiv Q(\overline{x}) \wedge SUC(\overline{x},\overline{x_{ini}});$
$P_2(\overline{x},\overline{x_{ini}},\overline{y}) \equiv P_1(\overline{x},\overline{x_{ini}}) \wedge R(\overline{x},\overline{y});$
$SUC(\overline{x},\overline{x_{ini}}) \equiv$ *ver en libro texto*

Ejemplo 1.- Transformación recursiva final a iterativa

En nuestro ejemplo1 una solución recursiva final es:

```
{Q ≡ 0<=i<=n  ^  r = Σα∈{1..i}. v[α]}
fun isuma2(v:vect[1..N] de entero; i:nat; r:entero) dev s:entero
        caso   i=n    → r
         []    i>0    → isuma2(v, i+1,r+v[i+1])
        fcaso
ffun
{R ≡ s = Σα∈{1..n}. v[α]}
```

y la llamada inicial era *isuma2(v, 0, 0)*

Su equivalente iterativo dará lugar a un bucle, (uno solo por ser final), en el que:

a) La inicialización de variables consistirá en asignar a las variables involucradas los mismos valores que se utilizan en la llamada inicial en la función recursiva:

   Inicialización $\equiv$ *<i, r>:= <0, 0>*

b) La condición de bucle será el caso no trivial en el algoritmo recursivo: $i < n$

c) Se devolverá tras el bucle el mismo valor que devuelve en el caso trivial el algoritmo recursivo: r

d) El invariante será el predicado dado por la igualdad entre la conjunción de la precondición del algoritmo recursivo y de la función aplicada a los valores iniciales, y la propia función para valores genéricos, es decir: $P(\overline{x}, \overline{x_{ini}}) \equiv Q(\overline{x}) \wedge f(\overline{x_{ini}}) = f(\overline{x})$
   en este caso:
   $P(r, i, 0, 0) \equiv (0 \leq i \leq n) \wedge r = \sum \alpha \in \{1..i\}.\ v[\alpha] \wedge isuma2(v, 0, 0) = isuma2(v, i, r)$

   En la práctica esto conduce al predicado dado por la precondición de la función recursiva ya que la postcondición no depende de los parámetros, luego:

   $P \equiv (0<=i<=n) \wedge (r = \sum \alpha \in \{1..i\}.\ v[\alpha])$

e) La precondición de la función iterativa será el predicado utilizado en la precondición de la recursiva en el que se sustituyen las variables genéricas por sus valores iniciales:

$$Q(\overline{xini}) = \left[(0 \leq i \leq n) \wedge r = \sum_{\alpha=1}^{i} v[\alpha]\right]_{0,0}^{i,r} \equiv (0 \leq 0 \leq n) \wedge \left(0 = \sum_{\alpha=1}^{0} v[\alpha]\right) \equiv cierto$$

El algoritmo iterativo será:

```
{Q ≡ cierto}
fun suma_iter1(v: vect[1..n] de ent) dev (s: ent)
   var  i: nat;  r: ent  fvar
   <i, r>:= <0, 0>;
   mientras  i < n  hacer    {P ≡ (0<=i<=n)  ^  (r = Σα∈{1..i}. v[α])}
         r:= r + v[i + 1];
         i:= i + 1;
   fmientras
   dev r
ffun
{R ≡ s = Σα∈{1..n}. v[α]}
```

Ejemplo 2.- Transformación recursiva final a iterativa

```
{Q ≡(1<=f<=n) ^ (0<=i<=m) ^ (bb = ∀α∈{1..i}. t[f,α]>0)}
fun ilinea2(t: vector[1..n, 1..m] de nat; f,i: nat; bb: bool) dev (b: bool)
        caso   i=m  →  bb
         []    i<m  →  ilinea2(t, f, i+1, bb ^ t[f,i+1]>0)
        fcaso
ffun
{R ≡ b = ∀α∈{1..m}. t[f,α]>0}
```

y la llamada inicial era: *ilinea2(t, f, 0, cierto)*

aplicando los mismos preceptos anteriores tendremos:

```
{Q ≡1<=f<=n }
fun ilinea2(t: vector[1..n, 1..m] de nat; f: nat) dev (b: bool)
   var  i: nat;  bb:bool  fvar;
   <i, bb>:= <0, cierto>;
   mientras  i < m  hacer  {P ≡(0<=i<=m) ^ (bb = ∀α∈{1..i}. t[f,α]>0)}
         <i, bb>:= <i+1, bb∧(t[f, i+1]>0)>
   fmientras
   dev bb
ffun
{R ≡ b = ∀α∈{1..m}. t[f,α]>0}
```

Ejemplo 1.- Transformación recursiva no final a iterativa

La solución recursiva no final del ejemplo 1 era:

```
{Q ≡0<=i<=N}
   fun isuma(v: vector[1..N] de enteros; i: natural) dev (s: entero)
        caso   i=0   →    0
         []    i>0   →    isuma(v, i-1) + v[i]
        fcaso
   ffun
{R ≡ s = Σα∈{1..i}. v[α]}
```

y la llamada inicial: *isuma(v, n)*

Su transformación a iterativa dará lugar a dos bucles

Antes del primer bucle se inicializan las variables con los mismos valores utilizados en la llamada inicial: *Inicialización ≡ i:= n*

El primer bucle consiste en realizar la operación de modificación de parámetros realizada en la llamada recursiva utilizando como condición de bucle la del caso no trivial de la función recursiva:

```
mientras  i > 0  hacer
        i:= i – 1
fmientras
```

(el bucle se utiliza para alcanzar el valor inicial de i (caso trivial) para iniciar el segundo bucle)

Antes del segundo bucle se ha de inicializar la variable que contenga finalmente el resultado con el valor del caso trivial de la función recursiva: *s:= 0*

(como hemos llevado el valor de i hasta 0 (caso trivial), s se inicializa con triv(x))

El segundo bucle consiste en aplicar sucesivamente:

a) la función inversa de la aplicada en la modificación de parámetros en la llamada recursiva: *i:= i + 1*

b) La operación adicional del algoritmo recursivo sobre las variables de salida y entrada (combinación): *s:= s + v[i]*

El bucle termina cuando las variables alcanzan de nuevo sus valores iniciales:

```
mientras  i ≠ n  hacer
    i:= i + 1;
    s:= s + v[i]
fmientras.
```

El algoritmo resultante será:

```
{Q ≡ cierto}
fun suma_iter2(v: vect[1..n] de ent) dev (s: ent)
   var  i: nat;  s: ent  fvar
   i:= n;
   mientras  i > 0  hacer  {P1}
         i:= i – 1;
   fmientras
s:= 0;
   mientras  i ≠ n  hacer   {P2}
         i:= i + 1;
         s:= s + v[i];
   fmientras
   dev s
ffun
{R ≡ s = Σ α ∈ {1..n}. v[α]}
```

En este ejemplo es evidente, como ya se ha comentado, que el primer bucle puede ser eliminado asignando simplemente a i el valor cero. Esto ocurrirá en todos los casos en que la operación de modificación de parámetros en la llamada recursiva sea muy simple.

En este caso existe la operación inversa de la aplicada en la modificación de parámetros en la llamada recursiva, pero esto no es siempre posible. En algunos casos la función inversa no puede expresarse por medio de una operación matemática. En estos casos los valores obtenidos en el primer bucle en cada operación de modificación han de ser acumulados en una estructura tipo pila. En el segundo bucle la operación inversa se sustituye por la extracción de esos valores desde la pila (ver manual Peña).

Queda por establecer P_1 y P_2, pero como de lo que uno no entiende es mejor no hablar dejo que cada uno busque su solución.

Ejemplo 2.- Transformación recursiva no final a iterativa

Partiendo de la función recursiva no final ilinea

```
{Q º(1<=f<=n)  ^  (0<=i<=m)}
fun ilinea(t: vector[1..n, 1..m] de nat; f,i: nat) dev (b: bool)
        caso    i=0  →  cierto
         []     i>0  →  ilinea(t, f, i-1)  ^  (t[f,i]>0)
        fcaso
ffun
{R º b =  "a Î {1..i}. t[f,a]>0}
```

cuya llamada inicial era: *ilinea(t, f, m)*

Su transformación a iterativo será:

```
{Q º 1<=f<=n }
fun ilinea_it2(t: vector[1..n, 1..m] de nat; f: nat) dev (b: bool)
   var   i: nat;   fvar;
   i:= m;
   mientras   i > 0   hacer     {P1}
          i:= i – 1
   fmientras
   b:= cierto
   mientras   i ¹ m   hacer       {P2}
          <i, b>:= <i+1, bÙ(t[f, i]>0)>
   fmientras
   dev b
ffun
{R º b =  "a Î {1..m}. t[f,a]>0}
```

Como siempre, falta determinar P1 y P2 (suerte)

EJEMPLOS Y EJERCICIOS:

De la ***colección de problemas curso 1999-2000*** conviene hacer los ejercicios **10.1, 10.2, 10.3, 11.3**

Ejercicios de transformación recursivo-iterativo de exámenes de años anteriores

EFICIENCIA DE ALGORITMOS

Pues efectivamente, yo os recomiendo dejar para el final este tema que es el primero de la asignatura.

Es muy recomendable que antes de abordar el tema en el libro de texto visitéis la página de Jerónimo Quesada y estudies sus apuntes sobre el tema.

En este tema no tengo nada que aportar, solo os recomendaría que os aprendáis las fórmulas ya que siempre caen cuestiones relacionadas con este tema en la parte de test, además de que en el estudio de los algoritmos también os pueden pedir el coste.

EJEMPLOS Y EJERCICIOS:

De la ***colección de problemas curso 1999-2000*** conviene hacer los ejercicios **5.1, 5.2, 5.3, 5.4**

Exámenes de años anteriores.

Espero que estas orientaciones os sirva de ayuda para el estudio de la asignatura.
Mucha suerte.

José Manuel.