ANOTACIÓN DE PROGRAMAS

Un programa GCL puede anotarse con comentarios que lo especifiquen. Verificar el programa es constatar, vía un cálculo de corrección como el de Hoare (o el de precondiciones más débiles, de Dijkstra), que el programa es correcto con respecto a la especificación.

ANOTACIÓN DE INSTRUCCIONES

Anotar un programa es proporcionar información que describa su semántica, con *aserciones* (predicados sobre el estado). Cada aserción describe una condición que debe cumplirse cuando la ejecución llegue al punto en que está anotada.

Un programa S debe anotarse, como mínimo, con una *precondición* Q y una *poscondición* R:

$$\{Q\}\ S\ \{R\}.$$

La anotación de un programa debe ser suficiente para entender su funcionamiento y, si se es estricto, para demostrar su corrección con respecto a la especificación. En la práctica se verá cuándo es necesario incluir anotaciones, de modo que se incluyan las "no evidentes" y se omitan las "evidentes". Lo evidente o no de una anotación tiene que ver con el sentido común de quien programa.

Por ejemplo, se puede anotar:

```
{Q: x=A ∧ y=B}
   t:= x;
{Q1: t=A ∧ x=A ∧ y=B}
   x:= y;
{Q2: t=A ∧ x=B ∧ y=B}
   y:= t;
{R: x=B ∧ y=A}
```

Pero, si ya se sabe que esta es una forma estándar de intercambiar valores de variables, uno aceptaría mejor (¡se entiende sin la anotación!):

```
{Q: x=A ∧ y=B}
   t:= x;
   x:= y;
   y:= t;
{R: x=B ∧ y=A}
```

También, cuando se tiene que describir una instrucción iterativa **do … od**, debería ser preciso indicar un invariante y una función cota. La función cota se puede omitir en ocasiones evidentes (v.gr., si se puede argumentar la terminación de alguna manera sencilla) e incluso el invariante se podría omitir, cuando la tarea es fácilmente comprensible.

Por ejemplo, considérese el siguiente código que manipula un arreglo `b[0..n-1] of nat`:

```
{Q: true}
k:= 0;
{Inv P: 0≤k≤n ∧ (∀i| 0≤i<k : b[i]=0)}
{Cota t: n-k}
do k≠n →    b[k]:= 0;
            k:= k+1
od
{R: (∀i| 0≤i<n : b[i]=0)}
```

Parecería suficiente escribir:

```
{Q: true}
k:= 0;
do k≠n →    b[k]:= 0;
            k:= k+1
od
{R: (∀i| 0≤i<n : b[i]=0)}
```

E incluso, entendiendo que apenas se está describiendo el algoritmo:

```
{Q: true}
b:= 0       // un abuso de notación, pero se entiende …
{R: (∀i| 0≤i<n : b[i]=0)}
```

CONTEXTOS

En el desarrollo de programas es usual encontrar condiciones que se deben cumplir antes, durante y al final de la ejecución del programa. Por ejemplo, si se tiene un arreglo de números del que hay que averiguar el máximo elemento, el arreglo no debería cambiar en ningún momento; si se tuviera que ordenar el arreglo, éste cambiaría, pero no los valores que contiene.

Un *contexto* es una condición sobre el estado de una instrucción que se cumple antes, durante y después de la ejecución de la instrucción. Para considerar contextos, se usa la notación:

```
[  Ctx C
   {Q}
    S
   {R}
]
```

Debe entenderse que `C` es una condición que se exige además de las anotadas dentro de `S`. En particular, debe valer que

```
{Q ∧ C}
S
{R ∧ C}
```

ESPECIFICACIÓN DE PROGRAMAS

En general, un programa debe explicar al establecer un contexto, una precondición y una poscondición. Se usan notaciones como:

```
Ctx C: …
Pre Q: …
Pos R: …
```

Si alguna de ellas se omite, se entiende que la condición correspondiente es `true`.

Las declaraciones de variables, si bien no son fórmulas lógicas, tienen el carácter de permanecer a lo largo del programa y, en este sentido, son contextuales al programa.

EJERCICIOS

1 *Coffee can problem*

"Una bolsa tiene B bolas blancas y N negras, y no está vacía. El siguiente proceso se repite siempre que se pueda:

- extraer dos bolas
- si son del mismo color, se tiran, pero se entra una negra a la bolsa
- si son de diferente color, se devuelve la blanca y se tira la negra"

Se puede probar que el proceso termina con una bola y ésta será negra, si y solo si el número inicial de bolas blancas es par.

Un algoritmo que simula el proceso:

```
[Ctx C: true
{Pre Q: 0≤b=B ∧ 0≤n=N ∧ b+n>0}
{Inv P: 0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B}
{Cota t: b+n}
      do b≥2 ∧ n≥0      →    b,n:= b-2,n+1
      [] b≥0 ∧ n≥2      →    n:= n-1
      [] b≥1 ∧ n≥1      →    n:= n-1
      od
{Pos R: (par.B ∧ b=0 ∧ n=1) ∨ (¬par.B ∧ b=1 ∧ n=0)}
]
```

2 *Ordenamiento por máximos*

Dado un arreglo de enteros, `b[0..n-1] of int`, ordenar sus elementos en el mismo arreglo. Una solución es usar un ordenamiento por máximos, explicado así:

```
[ Ctx C: perm(b,B)
  {Q: b = B}

  i:= n-1;

  {Inv P: 0≤i≤n-1 ∧ ord(b[i+1..n-1]) ∧ (i=n-1 ∨ (∀k| 0≤k≤i : b[k]<b[i+1]))}
  {Cota t: }

   do i≠0   →    j,jmax:= i-1,i;

                 {Inv P1: -1≤j<i  ∧  b[jmax] = (max k| j<k≤i : b[k])}

                 do j≠-1 → if b[jmax]<b[j] → jmax:= j
                           [] b[jmax]≥b[j] → skip
                           fi;
                           j:= j+1
                 od;

                 b[j],b[jmax],i:= b[jmax],b[j],i-1
   od

   {R: ord(b)}
]
```