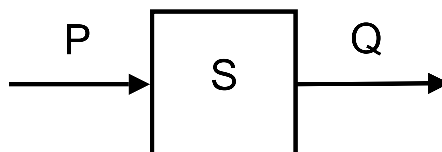
TEMA 6.- VERIFICACION DE PROGRAMAS IMPERATIVOS

6.1.- Lógica de Hoare.

- Desarrollada por C.A.R. Hoare (1969), es una lógica que permite probar la verdad o falsedad de propiedades de programas imperativos (especialmente corrección y terminación) sin concurrencia o paralelismo.
- Basada en la idea de diagrama de flujo anotado.



  - S es una "frase" de código en un programa de alto nivel, con una única entrada y una única salida normal.
    - Instrucción.
    - Bloque de instrucciones consecutivas.
    - Un programa o subprograma.
  - Cada frase S denota una relación entre dos estados en un espacio de estados definido por el producto cartesiano de los valores de las variables del programa. Es decir, al ejecutar una instrucción o secuencia de instrucciones S, se modifica el valor de una o varias variables, transitando de un estado a otro.
  - P y Q son *anotaciones*, fórmulas de la Lógica de Primer Orden (Lógica de Predicados).
    - P ≡ Precondición (se refiere al estado previo a S)
    - Q ≡ Postcondición (se refiere al estado posterior a S)

- Notación : $\{P\}S\{Q\}$
- Significado:
  - *Corrección parcial*: si la precondición P es cierta antes de ejecutar la frase de programa S, **y** la ejecución termina normalmente, la postcondición Q es cierta.

$$\models_{par} \{P\}S\{Q\}$$

  - *Corrección total*: si la precondición P es cierta antes de la ejecución de la frase de programa S, esta **termina** y la postcondición Q es cierta.

$$\models_{tot} \{P\}S\{Q\}$$

- En la corrección parcial, la frase S debe terminar **normalmente**. Existen dos posibles fuentes de terminación anormal:
  - Bucle infinito.
  - Error de ejecución (división entre cero, overflow, ...).
- Correccion total implica corrección parcial. Sin embargo, en muchas ocasiones se demuestra primero la segunda y, a partir de ella la primera.
- En Lógica de Hoare se utilizan dos lenguajes formales:
  - Un lenguaje de programación imperativo (LP) para S.
  - Un lenguaje de fórmula (LF) para P y Q. En LF se usan los tipos (enteros, booleanos, reales, ...), funciones (suma de enteros, suma de reales...) y predicados($<, >, \leq, \ldots$) usados en el LP, incluídos los definidos por el usuario.
- Para hacer pruebas sobre Lógica de Hoare presentaremos un método de prueba sintáctico. Primero veremos como demostrar si una tripla $\{P\}S\{Q\}$ es derivable para la condición de corrección parcial

  $$\vdash_{par} \{P\}S\{Q\}$$

  y después presentaremos las condiciones necesarias para tener corrección total.

  $$\vdash_{tot} \{P\}S\{Q\}$$

- Puede demostrarse (aunque no lo haremos, podeis respirad tranquilos) la coherencia de las pruebas, mientras que la complección se cumplirá sólo en determinadas circunstancias, tanto con corrección parcial como total.

  $$\text{coherencia} \quad \vdash_{par} \{P\}S\{Q\} \;\Rightarrow\; \models_{par} \{P\}S\{Q\}$$

  $$\text{compleccion} \quad \models_{par} \{P\}S\{Q\} \;\not\Rightarrow\; \vdash_{par} \{P\}S\{Q\}$$

- Sea S un programa y P su precondición, $\{P\}S\{Falso\}$ significa que S nunca termina cuando se comienza en P. Se trata de un problema indecidible (no se puede demostrar siempre), de modo que la lógica de Hoare no es completa.

6.1.1.- Reglas para la corrección parcial.

- *Asignación*:

$$\overline{\{P[E/x]\}\ x = E\ \{P\}}$$

  - Si el estado inicial es s, entonces el estado s' después de la asignación es igual a s, sustituyendo la variable x por el resultado de evaluar E.
  - Si s' satisface P (la hace cierta), la fórmula que satisfará s será $P[E/x]$, el resultado de sustituír todas las ocurrencias libres de x en P por E.
    Ejs.- $\{y + 5 = 10\}\ y = y + 5\ \{y = 10\}$
    $\{y + y < z\}\ x = y\ \{x + y < z\}$
    $\{2 * (y + 5) > 20\}\ y = 2 * (y + 5)\ \{y > 20\}$

- *Composición*:

$$\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$$

  Aplica la transitividad sobre dos frases para poder obtener la pre y postcondiciones de la concatenación de ambas.

- *Instrucción if*:

$$\frac{\{P \wedge B\}S_1\{Q\} \quad \{P \wedge \neg B\}S_2\{Q\}}{\{P\}\text{if } B \text{ then } S_1 \text{ else } S_2\{Q\}}$$

  - Si ambos bloques S_1 y S_2 tienen la misma postcondición Q, mientras que la conjunción de la fórmula P con la condición B y, respectivamente, con su negación $\neg B$, son sus precondiciones, P aparece como precondición de la bifurcación y Q como su postcondición.
  - Las condiciones del LP (normalmente expresiones booleanas) deben ser importadas como parte de las fórmulas lógicas de las anotaciones (en este caso precondiciones).

- *While parcial*:

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}\text{while } B \text{ do } S\{P \wedge \neg B\}}$$

  - Se trata de la regla que permite calcular las pre y post condiciones de un bucle while satisfaciendo la propiedad de corrección parcial.
  - Si la conjunción de las condiciones P y B es la precondición de S y P es su postcondición, quiere decir que cuando P y B son ciertas, P será cierto tras la ejecución de S. Por lo tanto, si B es la ondición del bucle while podemos demostrar que, siendo cierta la precondición P, se cumple la postcondición $P \wedge \neg B$, dado que P era postcondición de la(s) instruccion(es) S del interior del bucle y $\neg B$ debe cumplirse para salir del while.

- *Reforzamiento de la precondición (implicación izda.)*:

$$\frac{R \to P \quad \{P\}S\{Q\}}{\{R\}S\{Q\}}$$

  - Esta regla permite asumir una precondición más fuerte que la existente, de ser necesario, sin perder la corrección de la demostración del programa anotado.
  - Permite importar pruebas de la lógica de predicados (concretamente $R \to P$).

- *Debilitamiento de la postcondición (implicación dcha.)*:

$$\frac{\{P\}S\{Q\} \quad Q \to R}{\{P\}S\{R\}}$$

  - Esta regla permite asumir una postcondición más débil que la existente, de ser necesario, sin perder la corrección de la demostración del programa anotado.
  - Permite importar pruebas de la lógica de predicados (concretamente $Q \to R$).

- *Otras*:

$$(\text{conjunción}) \ \frac{\{P\}S\{Q_1\} \quad \{P\}S\{Q_2\}}{\{P\}S\{Q_1 \wedge Q_2\}} \qquad (\text{disyunción}) \ \frac{\{P_1\}S\{Q\} \quad \{P_2\}S\{Q\}}{\{P_1 \vee P_2\}S\{Q\}}$$

$$\begin{pmatrix}\text{conjunción}\\ \text{inversa}\end{pmatrix} \ \frac{\{P\}S\{Q_1 \wedge Q_2\}}{\{P\}S\{Q_i\}} \qquad \begin{pmatrix}\text{disyunción}\\ \text{inversa}\end{pmatrix} \ \frac{\{P_1 \vee P_2\}S\{Q\}}{\{P_i\}S\{Q\}} \quad i \in \{1,2\}$$

6.1.2.- Corrección parcial.

- Partiendo de una especificación (una postcondición y posiblemente alguna precondición), se trata de demostrar, utilizando las reglas anteriores (y algunas otras que no hemos visto) que el programa verifica dicha especificación.
- Generalmente, tras unas pocas líneas de código, la prueba se hace demasiado compleja para poder representarla en forma de inferencias encadenadas. Para resolver este problema se intercalan las anotaciones del LF entre las líneas del LP.
- Dado $S \equiv S_1; S_2; \ldots; S_n$, si queremos demostrar que $\{P_0\}S\{P_n\}$, basta demostrar que:

$$\{P_0\}S_1\{P_1\} \quad \{P_1\}S_2\{P_2\} \quad \ldots \quad \{P_{n-1}\}S_n\{P_n\}$$

y usar la regla de la composición $n-1$ veces.

De este modo, podemos representar la prueba de $\{P_0\}S\{P_n\}$ como:

$$\begin{array}{ll} \{P_0\} & \text{anotación} \\ S_1 & \\ \{P_1\} & \text{anotación} \\ S_2 & \\ \dots & \\ \{P_{n-1}\} & \text{anotación} \\ S_n & \\ \{P_n\} & \text{anotación} \end{array}$$

- Las fórmulas $P_1, \dots, P_{n-1}$ serán condiciones intermedias y cada paso

$$\begin{array}{l} \{P_{i-1}\} \\ S_i \\ \{P_i\} \end{array}$$

  será obtenido a través de alguna de las reglas anteriores.

- El proceso de prueba para un bloque o programa $S \equiv S_1; S_2; \dots ; S_n$ se produce de abajo a arriba (desde el fin hacia el principio del bloque o programa) debido a la naturaleza de la regla de la asignación. En general, comenzamos con la postcondición P_n y se utiliza S_n para obtener P_{n-1}.

- Obtener P_{i-1} a partir de P_i y S_i es mecánico para las asignaciones y las instrucciones if. La P_{i-1} obtenida de esa manera se denomina la *precondición más débil* para S_i y P_i, lo que quiere decir que que P_{i-1} es la fórmula lógica más débil que siendo verdadera al principio de la ejecución de S_i garantiza la veracidad de la postcondición P_i.

- Al final del proceso, obtenemos una fórmula P'_0 al principio de la secuencia, que garantiza que al ejecutar S se verifica la postcondición P_n. Debemos chequear si P'_0 puede obtenerse a partir de la precondición P_0 mediante la regla de reforzamiento de la precondición.

  En este caso, deberíamos demostrar que $P_0 \rightarrow P'_0$, usando cualquiera de los sistemas deductivos de lógica de predicados. En general, ese tipo de demostraciones, que aparecen al usar las dos reglas de la implicación, suelen omitirse en la secuencia de anotaciones e instrucciones. Si se demuestra que $P \rightarrow R$, la práctica habitual es simplemente escribir la segunda condición debajo de la primera.

$$\begin{array}{l} \{P\} \\ \{R\} \\ S \\ \{Q\} \end{array}$$

- Para la calcular la precondición más débil P para un if a partir de una prostcondición Q

$$\{P\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \ \{Q\}$$

suele precederse del siguiente modo:

1. Se obtiene la precondición P_1 a partir de Q y S_1.
2. Se obtiene la precondición P_2 a partir de Q y S_2.
3. $P \equiv (B \rightarrow P_1) \wedge (\neg B \rightarrow P_2)$

- Recordemos el while parcial:

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}\text{while } B \text{ do } S\{P \wedge \neg B\}}$$

- Si tenemos una precondición R y una postcondición Q, y queremos demostrar:

$$\models_{par} \{R\}\text{while } B \text{ do } S\{Q\}$$

Debemos encontrar una fórmula P del LF que verifique:

1. $R \rightarrow P$ (para el reforzamiento de la precondición)
2. $P \wedge \neg B \rightarrow Q$ (para el debilitamiento de la postcondición)
3. $\models_{par} \{P\}\text{while } B \text{ do } S\{P \wedge \neg B\}$ (regla while parcial)

- A la fórmula P verificando lo anterior se le denomina *invariante.*

- Encontrar invariantes requiere cierto ingenio (puede haber muchas diferentes para un mismo bucle), aunque hay reglas heurísticas:

1. Comenzar modificando la postcondición del while para hacerla dependiente del índice del bucle (la variable que crece o decrece en cada iteración).
2. Si la invariante P aún no verifica $P \wedge \neg B \rightarrow Q$, debemos reforzar P para que se cumpla.

6.1.3.- Corrección total.

- Lo anterior sólo prueba la corrección parcial de las triplas $\{P\}S\{Q\}$, es decir, sólo es cierto cuando S termina normalmente.

- Habíamos visto dos posibles razones para la no terminación: error de ejecución y bucle infinito. La primera está cubierta por el sistema de cálculo que ya hemos descrito.

- Por lo tanto, asegurar la corrección total implica asegurar la corrección parcial **y** la terminación de los bucles while.
- Podemos obtener la prueba de terminación si podemos, a partir de las variables de los estados del bucle, encontrar una expresión entera que disminuya en cada iteración del bucle manteniéndose no negativa (0 o positiva).
- Si tal expresión existe, en bucle termina después de un número finito de iteraciones, ya que no hay cadenas descendentes infinitas de enteros positivos (álgebra básica).
- A la expresión entera mencionada se le denomina *variante*.
- De este modo obtenemos una regla del *while total*:

$$\frac{\{P \wedge B \wedge (0 \leq E = E_0)\}\ S\ \{P \wedge (0 \leq E < E_0)\}}{\{P \wedge (0 \leq E)\}\ \text{while}\ B\ \text{do}\ S\ \{P \wedge \neg B\}}$$

  E es la variante, que decrece en cada iteración. Si $E = E_0$ antes de cada iteración del bucle, es estrictamente menor después. Si existen instrucciones antes del bucle y queremos calcular sus anotaciones a partir de la precondición del while, debemos utilizar la precondición del while.

  La aplicación de esta regla es equivalente a la utilización de la regla del while parcial y la demostración de las siguientes fórmulas:

$$P \wedge B \rightarrow E > 0 \quad \text{y} \quad \{P \wedge B \wedge E = E_0\}S\{E < E_0\}$$

6.1.4.- Complección relativa.

- Para una fórmula P y una frase S, sea $post(P, S)$, la fórmula más fuerte para la cual $\{P\}S\{post(P, S)\}$, tenemos que:

$$\text{Si } \{P\}S\{Q\} \text{ entonces } post(P, S) \rightarrow Q$$

- Para una fórmula Q y una frase S, sea $pre(S, Q)$, la fórmula más débil para la cual $\{pre(S, Q)\}S\{Q\}$, tenemos que:

$$\text{Si } \{P\}S\{Q\} \text{ entonces } P \rightarrow pre(S, Q)$$

- Supongamos que:
  1. $post(P, S)$ existe para cada P y S.
  2. $pre(S, Q)$ existe para cada S y Q.
  3. Las implicaciones $post(P, S) \rightarrow Q$ o $P \rightarrow pre(S, Q)$ pueden demostrarse.

Entonces cualquier tripla $\{P\}S\{Q\}$ podría ser demostrada, es decir, se verificaría la propiedad de complección:

$$\models_{par} \{P\}S\{Q\} \Rightarrow \vdash_{par} \{P\}S\{Q\}$$

- Pero sabemos que no es el caso. Por lo tanto, la no complección de la lógica de Hoare no surge de ella misma, sino de la indecidibilidad de la lógica de predicados más los axiomas de la aritmética (teorema de Gödel).

6.1.5.- Arrays y Procedimientos

- El uso de arrays en anotaciones presenta complicaciones extra, debido al hecho de que las condiciones involucran tanto a los índices como al contenido de los arrays. En el siguiente ejemplo podemos ver como una tripla supuestamente correcta no lo es:

$$\{x[1] = 1 \wedge x[2] = 3\}x[x[1]] = 2\{x[x[1]] = 2\}$$

  Al cambiar el valor de $x[1]$ la asignación cambia también el valor de $x[x[1]]$.

- La solución es hacer una regla análoga a la de la asignación pero tratando a los arrays como elementos completos.

- Para ello, primero hacemos la definición de la asignación en arrays. El array resultante de hacer la asignación $x[e_1] = e_2$ se define como:

$$(x; e_1 : e_2)[e_3] = \begin{cases} e_2 & \text{si } e_1 = e_3 \\ x[e_3] & \text{en otro caso} \end{cases}$$

- *Regla de asignación sobre arrays*

$$\frac{}{\{P[(x; e_1 : e_2)/x]\}\ x[e_1] = e_2\ \{P\}}$$

- En cuanto a las llamadas a procedimientos, si estos son no recursivos, las precondiciones y postcondiciones deducidas para el código del procedimiento son heredadas en cada llamada.

  Sea $h(x_1, \ldots, x_m; y_1, \ldots, y_n) = S$, donde x_i son las variables libres en P y asignadas en en el código S del procedimiento, mientras que y_j son las variables libres no asignadas en S.

$$\frac{\{P\}\ S\ \{Q\}}{\{P\}\ h(x_1, \ldots, x_m; y_1, \ldots, y_n)\ \{Q\}}$$

- En el caso de procedimientos recursivos, para demostrar la **corrección parcial**, podemos utilizar la precondición y postcondición esperadas para el procedimiento en la demostración. Esto es, definimos tales condiciones previamente a la prueba de corrección, utilizándolas en dicha prueba para las llamadas recursivas.

$$\frac{\begin{array}{c}\{P\}\ h(x_1,\dots,x_m;y_1,\dots,y_n)\ \{Q\}\\ \vdots\\ \{P\}\ S\ \{Q\}\end{array}}{\{P\}\ h(x_1,\dots,x_m;y_1,\dots,y_n)\ \{Q\}}$$

6.1.6.- Ejemplos

I.- División Entera

$\{x \geq 0 \wedge y \geq 0\}$
$a = 0;$
$b = x;$
while $b \geq y$ do
  $b = b - y;$
  $a = a + 1;$
od
$\{x = a * y + b \wedge b \geq 0 \wedge b < y\}$

1) El programa consta de dos asignaciones y un bucle. Calcularemos las anotaciones de abajo a arriba utilizando las reglas de la asignación y de la composición. Por lo tanto, el primer paso será calcular la invariante del bucle. Se trata de una fórmula P que verifica:

   $$\{P\}\text{while } B \text{ do } S\{P \wedge \neg B\}$$

   Una buena heurística es partir de la postcondición del programa. Así, la invariante podría ser $x = a * y + b \wedge b \geq 0$.

2) Vamos a intentar demostrar dicha invariante. Para ello, con el objetivo de poder utilizar la regla del while parcial, intentamos demostrar $\{P \wedge B\}S\{P\}$ donde P es la invariante y B es la condición del bucle. El primer paso es aplicar la regla de la asignación:

   $$\{x = (a+1) * y + b \wedge b \geq 0\}\ a = a + 1;\ \{x = a * y + b \wedge b \geq 0\}$$

3) Seguimos aplicando la asignación a la instrucción inmediatamente anterior:

   $$\{x = (a+1) * y + b - y \wedge b - y \geq 0\}\ b = b - y;\ \{x = (a+1) * y + b \wedge b \geq 0\}$$

4) Composición de 2) y 3):

$$\{x = (a+1)*y+b-y \wedge b-y \geq 0\}\ b = b-y;\ a = a+1;\ \{x = a*y+b \wedge b \geq 0\}$$

5) Utilizando las reglas del while parcial y del reforzamiento de la precondición, podemos demostrar que $\{P \wedge B\}S\{P\}$ si demostramos que:

$$(x = a*y+b \wedge b \geq 0 \wedge b \geq y) \rightarrow (x = (a+1)*y+b-y \wedge b-y \geq 0)$$

lo que es fácil si tenemos en cuenta que si $b \geq y$ entonces $b - y \geq 0$ y, además:

$$x = (a+1)*y+b-y = a*y+y+b-y = a*y+b$$

6) Ahora estamos en condiciones de utilizar la regla del while parcial para demostrar:

$\{x = a*y+b \wedge b \geq 0\}$
while $b \geq y$ do
  $b = b - y;$
  $a = a + 1;$
od
$\{x = a*y+b \wedge b \geq 0 \wedge b < y\}$

7) Partiendo de la invariante, podemos seguir ascendiendo en el programa usando la regla de la asignación:

$$\{x = a*y+x \wedge x \geq 0\}\ b = x;\ \{x = a*y+b \wedge b \geq 0\}$$

8) Aplicamos la regla de la asignación otra vez:

$$\{x = 0*y+x \wedge x \geq 0\}\ a = 0;\ \{x = a*y+x \wedge x \geq 0\}$$

9) Composición sobre 7) y 8):

$$\{x = 0*y+x \wedge x \geq 0\}\ a = 0;\ b = x;\ \{x = a*y+b \wedge b \geq 0\}$$

10) Composición sobre 6) y 9):

$\{x = 0*y+x \wedge x \geq 0\}$
$a = 0;$
$b = x;$
while $b \geq y$ do
  $b = b - y;$
  $a = a + 1;$
od
$\{x = a*y+b \wedge b \geq 0 \wedge b < y\}$

11) Como la postcondición del bucle coincide con la del programa únicamente nos queda por demostrar la precondición gracias a la regla de reforzamiento de la precondición. En particular, debemos demostrar que:

$$(x \geq 0 \land y \geq 0) \rightarrow (x = 0 * y + x \land x \geq 0)\}$$

lo que resulta obvio. Con ello, habríamos demostrado la corrección parcial (debido a la regla que hemos utilizado en el while) de este programa. Demostrar la corrección total es imposible, a no ser que incorporemos $y > 0$ como precondición del programa.

- Al final, podemos resumir la demostración intercalando las anotaciones en el programa:

$\{x \geq 0 \land y \geq 0\}$
$\{x = 0 * y + x \land x \geq 0\}$
$a = 0;$
$\{x = a * y + x \land x \geq 0\}$
$b = x;$
$\{x = a * y + b \land b \geq 0\}$
while $b \geq y$ do
  $\{x = (a + 1) * y + b - y \land b - y \geq 0\}$
  $b = b - y;$
  $\{x = (a + 1) * y + b \land b \geq 0\}$
  $a = a + 1;$
  $\{x = a * y + b \land b \geq 0\}$
od
$\{x = a * y + b \land b \geq 0 \land b < y\}$

II.- Potencia de enteros

$\{m \geq 0 \land n > 0\}$ // con $n > 0$ evitamos el caso 0^0
$r = 1;$
$i = 0;$
while $i < m$ do
  $r = r * n;$
  $i = i + 1;$
od
$\{r = n^m\}$

1) De nuevo, el programa consta de dos asignaciones y un bucle. Calcularemos las anotaciones de abajo a arriba utilizando las reglas de la asignación y de la composición, de modo que el primer paso será calcular una invariante P,

verificando la regla del while parcial:

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}\text{while } B \text{ do } S\{P \wedge \neg B\}}$$

Una buena heurística es partir de la postcondición del programa, haciéndola depender del índice i del bucle. Teniendo en cuenta que i varía de 0 a m, una posibilidad es sustituír m por i en la postcondición. Así, sabemos que la invariante debe contener la condición $r = n^i$. Ahora bien, sabemos que al final la conjunción entre la invariante y la negación de la condición del bucle ($i \geq m$) deben implicar a la postcondición del programa para poder aplicar la regla de debilitamiento de la postcondición. Para poder demostrar esa implicación, debemos demostrar que $i = m$ a la salida del bucle. Dado que la negación de la condición del while (que forma parte de su postcondición) es $m \geq i$, debemos incluír la fórmula $i \leq m$ en la invariante del bucle. De este modo, la fórmula de la invariante sería.

$$\{r = n^i \wedge i \leq m\}$$

2) Además, para probar la corrección total, necesitaremos una variante que se mantenga positiva y pero decrezca en cada iteración del bucle. La elección más inmediata es $m - i$.

3) Primero vamos a intentar demostrar la corrección del bucle. Aplicamos la regla de la asignación sobre $\{P \wedge (0 \leq E < E_0)\}$ donde P es la postcondición y E es la variante.

$$\{r = n^{i+1} \wedge i + 1 \leq m \wedge 0 \leq m - (i + 1) < E_0\}$$
$$i = i + 1;$$
$$\{r = n^i \wedge i \leq m \wedge 0 \leq m - i < E_0\}$$

4) Aplicamos la regla de la asignación sobre la instrucción inmediatamente anterior:

$$\{r * n = n^{i+1} \wedge i + 1 \leq m \wedge 0 \leq m - (i + 1) < E_0\}$$
$$r = r * n;$$
$$\{r = n^{i+1} \wedge i + 1 \leq m \wedge 0 \leq m - (i + 1) < E_0\}$$

5) Composición de 3) y 4):

$$\{r * n = n^{i+1} \wedge i + 1 \leq m \wedge 0 \leq m - (i + 1) < E_0\}$$
$$r = r * n;$$
$$i = i + 1;$$
$$\{r = n^i \wedge i \leq m \wedge 0 \leq m - i < E_0\}$$

6) Debemos demostrar que que se cumple la precondición $\{P \wedge B \wedge (0 \leq E = E_0)\}$. Lo hacemos mediante la regla de reforzamiento de la precondición aplicada sobre la anotación que hemos calculado en 5). Para ello, debemos demostrar:

$$(r = n^i \wedge i \leq m \wedge i < m \wedge 0 \leq m - i = E_0) \rightarrow$$
$$(r * n = n^{i+1} \wedge i + 1 \leq m \wedge 0 \leq m - (i + 1) < E_0)$$

Lo cual es fácil si tenemos en cuenta que:

1. $r * n = n^i * n = n^{i+1}$, lo que puede demostrarse a partir de $r = n^i$.
2. $i + 1 \leq m$ se demuestra partiendo de $i < m$.
3. $0 \leq m - (i + 1) < E_0$ es inmediato a partir de $0 \leq m - i = E_0$ e $i < m$.

7) Ahora debemos propagar la precondición del bucle hacia las intrucciones precedentes aplicando la regla de la asignación:

$$\{r = n^0 \wedge 0 \leq m \wedge 0 \leq m\}\ i = 0;\ \{r = n^i \wedge i \leq m \wedge 0 \leq m - i\}$$

como $0 \leq m$ está repetida en la reescritura de la variante y en la de la invariante, la escribimos sólo una vez a partir de ahora.

8) Volvemos a aplicar la regla de asignación:

$$\{1 = n^0 \wedge 0 \leq m\}\ r = 1;\ \{r = n^0 \wedge 0 \leq m\}$$

9) Aplicando la composición sobre 7) y 8), obtenemos:

$$\{1 = n^0 \wedge 0 \leq m\}\ r = 1;\ i = 0;\ \{r = n^i \wedge i \leq m \wedge 0 \leq m - i\}$$

10) Aplicamos la composición sobre 6) y 9):

$$\{1 = n^0 \wedge 0 \leq m\}$$
$$Programa$$
$$\{r = n^i \wedge i \leq m \wedge i \geq m\}$$

11) Ahora debemos demostrar que la precondición del programa que fijamos al principio implica a la precondición que hemos obtenido en 10), para poder aplicar la regla de reforzamiento de la precondición. Hay que demostrar que:

$$(m \geq 0 \wedge n > 0) \rightarrow (1 = n^0 \wedge 0 \leq m)\}$$

lo que podría hacer hasta un chimpancé no muy listo.

12) Lo único que queda por demostrar es la postcondición del programa, y podemos hacerlo aplicando la regla de debilitamiento de la postcondición:

$$\frac{\{P\}S\{Q\} \quad Q \rightarrow R}{\{P\}S\{R\}}$$

donde Q sería la postcondición del while, es decir, la conjunción entre la invariante y la negación de la condición del bucle, y R la postcondición del programa. Para que el antecedente de la regla se cumpla, debemos demostrar que:

$$(r = n^i \wedge i \leq m \wedge i \geq m) \rightarrow (r = n^m)$$

A partir de $i \leq m$ y $i \geq m$, demostramos que $i = m$ y $r = n^m$.

- En la demostración anterior hemos optado por demostrar la corrección de la invariante y la variante del bucle while antes de construír las anotaciones de las instrucciones del programa y demostrar su concordancia con la precondición de este. Podría fácilmente, sin menoscabo de la corrección en la demostración, haber seguido el camino opuesto: verificar primero si la variante y la invariante de bucle permiten demostrar la precondición y postcondición del programa para comprobar después que son correctas en el bucle while.

- Otra opción para demostrar la corrección total en el bucle hubiera sido demostrar primero el cumplimiento de la regla del while parcial y asegurar la corrección total del bucle. En este caso, necesitaremos una invariante algo más reforzada:

$$r = n^i \wedge 0 \leq i \leq m$$

1) Propagamos la invariante de bucle hacia arriba mediante la regla de la asignación en cada una de las instrucciones del bucle:

$$\{r = n^{i+1} \wedge 0 \leq i+1 \leq m\}\ i = i+1;\ \{r = n^i \wedge 0 \leq i \leq m\}$$

$$\{r * n = n^{i+1} \wedge 0 \leq i+1 \leq m\}\ r = r * n;\ \{r = n^{i+1} \wedge 0 \leq i+1 \leq m\}$$

Y aplicamos la regla de la composición para obtener:

$$\{r * n = n^{i+1} \wedge 0 \leq i+1 \leq m\}\ r = r * n;\ i = i+1;\ \{r = n^i \wedge 0 \leq i \leq m\}$$

2) Demostraremos la corrección parcial utilizando la regla de reforzamiento de la precondición con $P \wedge B$ (siendo P la invariante y B la condición del bucle) y la precondición que hemos calculado para las dos instrucciones del bucle. Debemos ver si se verifica:

$$(r = n^i \wedge 0 \leq i \leq m \wedge i < m) \rightarrow (r * n = n^{i+1} \wedge 0 \leq i+1 \leq m)$$

lo que, en efecto, es cierto (ver paso 6 en la demostración anterior)

3) Para verificar la corrección total, debemos demostrar lo siguiente:

$$P \wedge B \rightarrow E > 0 \quad \text{y} \quad \{P \wedge B \wedge E = E_0\}\ S\ \{E < E_0\}$$

donde E es la variante y E_0 una constante entera positiva. La implicación, que si es cierta demuestra que la variante es positiva al entrar en el bucle, es fácil de demostrar:

$$(r = n^i \wedge 0 \leq i \leq m \wedge i < m) \rightarrow (m - i > 0)$$

dado que la condición del bucle nos dice que $i < m$, y la invariante que $0 \leq i \leq m$. Demostrar esta implicación es la razón por la cual necesitábamos reforzar la invariante. Sin saber que m e i son positivos, es imposible demostrar que $m - i$ también lo es.

4) En cuanto a $\{P \wedge B \wedge E = E_0\}\ S\ \{E < E_0\}$ significa que la variante se decrementa en cada iteración del bucle. Para demostrarlo, aplicamos la regla de la asignación en cada una de las instrucciones del bucle, más la regla de la composición.

$$\{m - (i + 1) < E_0\}\ r = r * n;\ i = i + 1;\ \{m - i < E_0\}$$

5) Para demostrar que $\{P \wedge B \wedge E = E_0\}$ es una precondición de las instrucciones del bucle, utilizamos la regla de reforzamiento de la precondición. Debemos verificar que:

$$(r = n^i \wedge 0 \leq i \leq m \wedge i < m \wedge m - i = E_0) \rightarrow (m - (i + 1) < E_0)$$

lo que resulta obvio: si $m - i = E_0$, entonces $m - (i + 1) < E_0$. Con ello habríamos demostrado la corrección total en el bucle.

III.- Máximo común divisor

$$
\begin{array}{l}
\{x_1 > 0 \wedge x_2 > 0\} \\
y_1 = x_1; \\
y_2 = x_2; \\
\text{while } y_1 \neq y_2 \text{ do} \\
\quad \text{if } y_1 > y_2 \text{ then } y_1 = y_1 - y_2; \\
\qquad\qquad\quad \text{else } y_2 = y_2 - y_1; \\
\text{od} \\
\{y_1 = mcd(x_1, x_2)\}
\end{array}
$$

1) En este caso no podemos utilizar la heurística de hacer la postcondición dependiente del índice del bucle para obtener la invariante. Para empezar, no hay índice del bucle. Necesitamos una expresión más o menos concordante

con la postcondición del programa pero que sea cierta en cada iteración del bucle. Una buena posibilidad es $mcd(y_1, y_2) = mcd(x_1, x_2)$. Como, además, tenemos intención de demostrar la corrección total y la variante más sencilla posible es $y_1 + y_2$, es buena idea incluír en la invariante $y_1 > 0 \wedge y_2 > 0$.

$$P = \{mcd(y_1, y_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 > 0\}$$
$$E = \{y_1 + y_2\}$$

2) Utilizaremos la regla del while total:

$$\frac{\{P \wedge B \wedge (0 \leq E = E_0)\}\ S\ \{P \wedge (0 \leq E < E_0)\}}{\{P \wedge (0 \leq E)\}\ \text{while } B \text{ do } S\ \{P \wedge \neg B\}}$$

La postcondición será:

$$\{mcd(y_1, y_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 > 0 \wedge 0 \leq y_1 + y_2 < E_0\}$$

Propagamos la variante a través de ambas opciones del bucle, usando la regla de la asignación:

$\{mcd(y_1 - y_2, y_2) = mcd(x_1, x_2) \wedge y_1 - y_2 > 0 \wedge y_2 > 0 \wedge 0 \leq y_1 < E_0\}$
$y_1 = y_1 - y_2;$
$\{mcd(y_1, y_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 > 0 \wedge 0 \leq y_1 + y_2 < E_0\}$

$\{mcd(y_1, y_2 - y_1) = mcd(x_1, x_2) \wedge y_2 - y_1 > 0 \wedge y_2 > 0 \wedge 0 \leq y_2 < E_0\}$
$y_2 = y_2 - y_1;$
$\{mcd(y_1, y_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 > 0 \wedge 0 \leq y_1 + y_2 < E_0\}$

3) Utilizamos la regla del if:

$$\frac{\{R \wedge B_{if}\}S_1\{Q\} \quad \{R \wedge \neg B_{if}\}S_2\{Q\}}{\{R\}\text{if } B_{if} \text{ then } S_1 \text{ else } S_2\{Q\}}$$

donde R será la precondición de las sentencias del bucle en la regla del while total: $\{P \wedge B \wedge (0 \leq E = E_0)\}$, siendo P la invariante y E la variante del bucle while. Utilizamos la regla del reforzamiento de la precondición, con lo que debemos demostrar dos implicaciones:

Para el then: $(P \wedge B \wedge (0 \leq E = E_0) \wedge B_{if}) \rightarrow$ Precondición para S_1

$$(mcd(y_1, y_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 > 0 \wedge y_1 \neq y_2 \wedge 0 \leq y_1 + y_2 = E_0 \wedge y_1 > y_2)$$
$$\rightarrow (mcd(y_1 - y_2, y_2) = mcd(x_1, x_2) \wedge y_1 - y_2 > 0 \wedge y_2 > 0 \wedge 0 \leq y_1 < E_0)$$

- $mcd(y_1 - y_2, y_2) = mcd(x_1, x_2)$ se demuestra a partir de $y_1 > y_2$ y $mcd(y_1, y_2) = mcd(x_1, x_2)$.

- $y_1 - y_2 > 0$ se demuestra teniendo en cuenta que $y_1 > 0$, $y_2 > 0$ y $y_1 < y_2$.
- $y_2 > 0$ está presente en al antecedente de la implicación.
- $0 \leq y_1 < E_0$ dado que $0 \leq y_1 + y_2 = E_0$ e $y_2 > 0$.

Para el else: $(P \wedge B \wedge (0 \leq E = E_0) \wedge \neg B_{if}) \rightarrow$ Precondición para S_2

$$(mcd(y_1, y_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 > 0 \wedge y_1 \neq y_2 \wedge 0 \leq y_1 + y_2 = E_0 \wedge y_1 \leq y_2)$$
$$\rightarrow (mcd(y_1, y_2 - y_1) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 - y_1 > 0 \wedge 0 \leq y_2 < E_0)$$

- $mcd(y_1, y_2 - y_1) = mcd(x_1, x_2)$ se demuestra a partir de $y_1 < y_2$ (dado que $y_1 \leq y_2$ e $y_1 \neq y_2$) y $mcd(y_1, y_2) = mcd(x_1, x_2)$.
- $y_2 - y_1 > 0$ se demuestra teniendo en cuenta que $y_1 > 0$, $y_2 > 0$ e $y_1 < y_2$, dado que $y_1 \leq y_2$ e $y_1 \neq y_2$.
- $y_1 > 0$ está presente en al antecedente de la implicación.
- $0 \leq y_2 < E_0$ dado que $0 \leq y_1 + y_2 = E_0$ e $y_1 > 0$.

4) Con lo anterior hemos demostrado el antecedente de la regla del while total. Propagamos la precondición en el consecuente $(P \wedge (0 \leq E))$ hacia arriba, siendo P la invariante y E la variante. Para ello, utilizaremos la regla de la asignación:

$$\{mcd(y_1, x_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge x_2 > 0 \wedge 0 \leq y_1 + x_2\}$$
$$y_2 = x_2;$$
$$\{mcd(y_1, y_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 > 0 \wedge 0 \leq y_1 + y_2\}$$

5) Otra vez la regla de la asignación:

$$\{mcd(x_1, x_2) = mcd(x_1, x_2) \wedge x_1 > 0 \wedge x_2 > 0 \wedge 0 \leq x_1 + x_2\}$$
$$y_1 = x_1;$$
$$\{mcd(y_1, x_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge x_2 > 0 \wedge 0 \leq y_1 + x_2\}$$

6) Y la regla de la composición:

$$\{mcd(x_1, x_2) = mcd(x_1, x_2) \wedge x_1 > 0 \wedge x_2 > 0 \wedge 0 \leq x_1 + x_2\}$$
$$y_1 = x_1;$$
$$y_2 = x_2;$$
$$\{mcd(y_1, y_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 > 0 \wedge 0 \leq y_1 + y_2\}$$

7) La regla de la composición otra vez, para 6) y 4).

$$\{mcd(x_1, x_2) = mcd(x_1, x_2) \wedge x_1 > 0 \wedge x_2 > 0 \wedge 0 \leq x_1 + x_2\}$$
$$Programa$$
$$\{mcd(y_1, y_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 > 0 \wedge y_1 = y_2\}$$

8) Demostramos la precondición del programa usando la regla de reforzamiento de la precondición:

$$(x_1 > 0 \wedge x_2 > 0) \rightarrow (mcd(x_1, x_2) = mcd(x_1, x_2) \wedge x_1 > 0 \wedge x_2 > 0 \wedge 0 \leq x_1 + x_2)$$

lo que resulta evidente.

9) Finalmente, demostramos la postcondición del programa a partir de la postcondición del while $\{P \wedge \neg B\}$, donde P es la invariante y B la condición del while,

$$(mcd(y_1, y_2) = mcd(x_1, x_2) \wedge y_1 > 0 \wedge y_2 > 0 \wedge y_1 = y_2) \rightarrow (y_1 = mcd(x_1, x_2))$$

fácilmente demostrable, dado que si $y_1 = y_2$ entonces $y_1 = mcd(y_1, y_2)$.

6.2.- La herramienta WHY

- Herramienta que demuestra la adecuación entre un programa con su especificación, basada en lógica de Hoare.
  - El LP es un dialecto del ML (una sintaxis próxima a Caml).
  - Uso de anotaciones para expresar la especificación.
  - La salida es un fichero con obligaciones de prueba (objetivos a demostrar) para varios demostradores de teoremas: Coq, PVS, Isabelle/HOL, HOL 4, HOL Light, Mizar, Simplify, haRVey y CVC Lite.
- El invocación en línea de comandos:

  ```
  why fichero.mlw
  ```

  da como resultado un fichero `foo_why.v` para el intérprete Coq, con los objetivos que deben demostrarse para garantizar la corrección del programa anotado.
- También se puede generar código Caml:

  ```
  why --ocaml fichero.mlw
  ```

  En este caso, no se generan obligaciones de prueba, sino que se genera código Caml correspondiente al programa en la salida estándas (o en un fichero con la opción `--output`).

  La opción `--ocaml-annot` permite preservar en el código Caml generado las anotaciones presentes en `fichero.mlw` en forma de comentarios.
- También se proporciona una herramienta para convertir ficheros WHY a HTML:

  ```
  why2html [-t titulo] fichero.mlw
  ```

  obteniendo como resultado un fichero HTML `fichero.mlw.html`.

6.2.1.- Lenguaje de programación

Los ficheros de entrada presentan la siguiente sintaxis:

- let x = programa_anotado

  Declara una expresión de programa con nombres. El programa es añadido al contexto actual y se generan las consiguientes en obligaciones de prueba.

  ```
  let c = 1+2
  let f = fun (x:int ref) -> x := !x + c
  ```

- parameter $x_1, \ldots, x_n : \tau$

  Declara parámetros que serán convertidos en axiomas en el demostrador de teoremas.

  ```
  parameter x,y : int ref
  parameter N : int
  parameter t : int array
  ```

  El prefijo `external` indica que $x_1, \ldots, x_n$ están ya definidas en el lado del demostrador.

  ```
  external parameter mean : int -> int -> int
  external parameter diff : a:int -> b:int ->
                            { b >= 0 } int { a = b + result }
  ```

- exception E [of τ]

  Declara una nueva excepción E con un argumento de tipo τ.

- logic $x_1, \ldots, x_n : \tau_1, \ldots, \tau_m - > \tau$

  Permite introducir los símbolo lógicos $x_1, \ldots, x_n$, que son predicados (si τ es `Prop`), constantes (si $m = 0$) o símbolos de funciones.

  ```
  logic max : int,int -> int
  logic is_int : real -> prop
  ```

  El prefijo `external` también puede ser usado, con el mismo significado que en `parameter`.

- predicate $p(x_1 : \tau_1, \ldots, x_n : \tau_n) = predicado$

  Define un predicado p.

  ```
  predicate ge0(x:int, y:int) = x >= y >= 0
  ```

- function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = term$

  Define una función lógica f.

  ```
  function f(x:int, y:int) : int = x + y
  ```

- axiom $x : predicado$

  Declara un axioma de nombre x.

  ```
  axiom max_1 : forall x:int. forall y:int. max(x,y) >= x
  ```

6.2.2.- Anotaciones

- La anotaciones de escriben con la sintaxis de lógica de predicados que es independiente el demostrador de teoremas utilizado. Se trata de predicados sobre los valores de las variables del programa visibles en la posición del programa en la que se escriben.

- Las pre y postcondiciones son escritas usando la sintaxis tradicional de la lógica de Hoare:

  $$\{predicado\}\ expresion\ \{predicado\}$$

  Dentro de una tripla, la pre o postcondiciones pueden ser omitidas. Dentro de una postcondición, el valor de la referencia `x` antes de la evaluación (en el punto de la precondición) es referenciado como `x@`.

  ```
  { } begin x := !x + 1 end { x > x@ }
  { x > 0 } begin x := 2 * !x; x := !x - 1 end { x > 0 }
  ```

  En la postcondición de unn programa o función el resultado está contenido en la variable `result`.

  ```
  { } 1 + 2 { result = 3 }
  { } begin x := !x + 1; !x end { result <> x@ }
  ```

- La precondición de un programa o cuerpo de función es una hipótesis que se deberá demostrar cierta cuando el programa o función son llamados, como `x=0` en:

  ```
  let f (u:unit) = { x = 0 } x := x + 1 { x = 1 }
  ```

  En cambio, las precondiciones más internas son *obligaciones de prueba*, que deberán ser demostradas en estado del programa correspondiente.

  ```
  let f (u:unit) =
    {} begin x := 0; { x = 0 } x := x + 1 {} end { x = 1 }
  ```

- No todas las anotaciones para las pre y postcondiciones tienen que ser introducidas en el código del programa, muchas de ellas son calculadas por WHY. Dentro de una secuencia de instrucciones, aquellas anotaciones necesarias pueden ser introducidas por el comando `assert`.

  ```
  begin
    x := 2 * !x;
    assert { even(x) };
    x := !x - 1;
  end
  ```

- Para probar la corrección de los bucles, se puede asociar a ellos tanto variantes como invariantes (las primeras son obligatorias y en muchos casos, será necesario incluír las segundas). La forma de especificar ambas condiciones será mediante una única anotación situada después de la palabra clave `do`, y las palabras reservadas `variant` e `invariant`.

  while *expresion* do { invariant *predicado* variant *termino*} *secuencia* done

  Por ejemplo:

  ```
  while !x > 0 do { invariant x >= 0 variant x }
    x := !x - 1
  done

  while !x < 10 do { invariant x <= 10 variant 10-x }
    x := !x + 1
  done
  ```

  Mientras que las invariantes son expresiones de la lógica de predicados, las variantes pueden ser de cualquier tipo y pueden tener asociadas cualquier relación de orden. Si no se especifican, el tipo entero y la relación de orden sobre los enteros se usarán por defecto.

- Las funciones recursivas son anotadas con el tipo del resultado y una variante:

  ```
  let rec f91 (n:int) : int { variant max(0,101-n) } =
    {}
    if n <= 100 then
      (f91 (f91 (n + 11)))
    else
      n - 10
    { (n <= 100 and result = 91) or
      (n >= 101 and result = n - 10) }
  ```

  Las variantes son como las utilizadas en los bucles.

- Se pueden insertar etiquetas en cualquier punto del programa usando la palabra clave `label`. Si se define una etiqueta `L`, el valor de la variable `x` en la posición del programa definida por `L`, será `x@L`.

  ```
  begin
    x := y;
    label L;
    { } begin x := !x + 1 end { x > x@L }
  end
  ```

La visibilidad de una etiqueta es la misma que la de una variable local.

- La construcción `absurd` se puede utilizar para denotar un punto del código inalcanzable.

```
{ 0 <= x <= 1 }
if x = 0 then
  ...
else if x = 1 then
  ...
else
  absurd
{ ... }
```

  En estos puntos, el usuario deberá demostrar que el contexto en, de hecho, absurdo, lo que se corresponde con la obligación de prueba `false`.

- Las expresiones del programa tiene asociadas tipos, incluyendo efectos colaterales y especificación. Este tipo es inferido por WHY para cada declaración `let`, o dado por el usuario en las declaraciones `parameter`. La sintaxis de estos tipos es la típica de los lenguajes funcionales, con sintaxis adicional para efectos colaterales y pre y postcondiciones.

  Tipos simples:

```
int
int -> int
int -> int ref -> unit
```

  Con pre y postcondiciones:

```
x:int -> y:int -> { } int { result = x + y }
```

  Con especificación de efecto colateral:

```
unit -> { x >= 0 } unit writes x { x < x@ }
a:int ref -> b:int ref -> { } unit reads b writes a { a = a@ + b }
n:int -> { } int raises Negative { result = sqrt(n) | Negative => n<0 }
```

  Cambiando el nombre del resultado:

```
x:int -> y:int -> { } returns z:int { z = x + y }
```

6.2.3.- Funciones predefinidas

Tanto para el LP como el LF.

- $+, -, *, /, \div$ para enteros.

```
add_int, sub_int, mul_int, div_int, mod_int : nat -> nat -> nat
```

- $-$ unario para enteros.

```
neg_int : int -> int
```

- $+, -, *, /$ para reales.

```
add_real, sub_real, mul_real, div_real : real -> real -> real
```

- $-$ unario y raíz cuadrada para reales.

```
neg_real, sqrt_real : real -> real
```

- Conversión de un entero en un número real.

```
real_of_int : int -> real
```

- Tamaño de un array.

```
array_length : array -> real
```

Exclusivamente para el lenguaje de programación

- $<, \leq, >, \geq, =, \neq$, entre enteros.

```
lt_int : x:int -> y:int ->
      {} bool {if result then x < y else x >= y}
le_int : x:int -> y:int ->
      {} bool {if result then x <= y else x > y}
gt_int : x:int -> y:int ->
      {} bool {if result then x > y else x <= y}
ge_int : x:int -> y:int ->
      {} bool {if result then x >= y else x < y}
eq_int : x:int -> y:int ->
      {} bool {if result then x = y else x <> y}
neq_int :  x:int -> y:int ->
      {} bool {if result then x <> y else x = y}
```

- $<, \leq, >, \geq, =, \neq$, entre reales, se definen con las mismas postcondiciones que para enteros.

```
lt_real, le_real, gt_real, ge_real, eq_real, neq_real
```

- $=, \neq$ para bool y unit, se definen con las mismas postcondiciones que para enteros y reales.

```
eq_bool, neq_bool, eq_unit, neq_unit
```

6.2.4.- Predicados predefinidos

- $<, \leq, >, \geq, =, \neq$, entre enteros.

```
lt_int, le_int, gt_int, ge_int, eq_int, neq_int:
                                    int -> int -> prop
```

- $<, \leq, >, \geq, =, \neq$, entre reales.

```
lt_real, le_real, gt_real, ge_real, eq_real, neq_real:
                                        real -> real -> prop
```

- $=, \neq$, entre booleanos.

```
eq_bool, neq_bool : bool -> bool -> prop
```

- $=, \neq$, entre unit.

```
eq_unit, neq_unit : unit -> unit -> prop
```

- $sorted_array(t, i, j)$ es cierto cuando el subarray $t[i..j]$ está ordenado de forma ascendente.

```
sorted_array : int array -> int -> int -> prop
```

- $exchange(t_1, t_2, i, j)$ es cierto cuando el array t_1 es igual al array t_2 intercambiando los elemantos con índice i y j.

```
exchange : int array -> int array -> int -> int -> prop
```

- *sub_permut*(t_1, t_2, i, j) es cierto cuando los subarrays $t_1[i..j]$ y $t_2[i..j]$ son permutaciones de uno en otro, siendo el resto de t_1 y t_2 iguales.

```
sub_permut : int -> int -> int array -> int array -> prop
```

- *permut*(t_1, t_2) es cierto cuando t_1 y t_2 son permutaciones de cada uno.

```
permut : int array -> int array -> prop
```

- *array_id*(t_1, t_2, i, j) es cierto cuando los subarrays $t_1[i..j]$ y $t_2[i..j]$ son idénticos.

```
array_id : int -> int -> int array -> int array -> prop
```