Clase teórica 11

Lógica de Hoare

Correctitud parcial y total de un programa

- **π(S, σ)** denota la computación de un programa S a partir de un estado inicial σ.

- **val(π(S, σ)) = σ´** denota el estado final de π(S, σ).
  En particular, **val(π(S, σ)) =** ⊥ denota que π(S, σ) no termina.

- Un programa S es **<u>correcto parcialmente</u>** con respecto a una especificación (p, q) sii:

  Para todo estado σ: $[\sigma \models p \ \wedge\ \mathbf{val(\pi(S, \sigma)) = \sigma' \neq \perp}] \ \rightarrow\ \sigma' \models q$

  es decir, a partir de un estado σ |= p, **si S termina (o no diverge)** lo hace en un estado σ´ |= q.

- Un programa S es **<u>correcto totalmente</u>** con respecto a una especificación (p, q) sii:

  Para todo estado σ: $\sigma \models p \ \rightarrow\ [\mathbf{val(\pi(S, \sigma)) = \sigma' \neq \perp} \ \wedge\ \sigma' \models q]$

  es decir, a partir de un estado σ |= p, **S termina (o no diverge)** lo hace en un estado σ´ |= q.

- **{p} S {q}** denota la correctitud parcial y **⟨p⟩ S ⟨q⟩** denota la correctitud total.

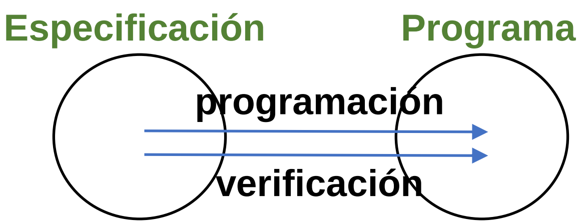
Las dos propiedades se distinguen porque se prueban con **métodos distintos**.

<u>Ejercicio</u> (asumiendo que *true* representa cualquier estado)

1. ¿Se cumple {x = 11} while x ≠ 0 do x := x – 2 od {true}?
2. ¿Se cumple ⟨x = 11⟩ while x ≠ 0 do x := x – 2 od ⟨true⟩?

Componentes de la Lógica de Hoare (repaso y ampliación)

1. Lenguaje de programación



- Instrucciones:

  S :: skip | x := e | S_1 ; S_2 | if B then S_1 else S_2 fi | while B do S_1 od

  Agregamos la instrucción skip, que no modifica el estado inicial (útil p.ej. para simular un if sin else)

- Expresiones de tipo entero:

  e :: n | x | $e_1 + e_2$ | $e_1 - e_2$ | $e_1 . e_2$ | …

  n es una constante entera. x es una variable entera.

- Expresiones de tipo booleano:

  B :: true | false | $e_1 = e_2$ | $e_1 < e_2$ | … | $\neg B_1$ | $B_1 \vee B_2$ | $B_1 \wedge B_2$ | …

Ejemplo de programa

```
a := 1 ; y := 1 ;
while a < x do
   a := a + 1 ; y := y . a
od
```

2. Lenguaje de especificación (lógica de predicados):

p :: true | false | $e_1 = e_2$ | $e_1 < e_2$ | … | $\neg p$ | $p_1 \vee p_2$ | … | $\exists x$: p | $\forall x$: p

Ejemplos de predicados

true

$x + 1 = y$

$\neg(a < x)$

$\forall x: (x > y \vee x \leq y)$

3.1. Axiomática para la correctitud parcial (Método H)

1. **Axioma del skip (SKIP)** $\{p\}\ \text{skip}\ \{p\}$

2. **Axioma de la asignación (ASI)** $\{p[x|e]\}\ x := e\ \{p\}$

   p[x|e] denota la sustitución en el predicado p de toda ocurrencia libre de la variable entera x libre por la expresión entera e

3. **Regla de la secuencia (SEC)**

$$\frac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

4. **Regla del condicional (COND)**

$$\frac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{q\}}$$

5. **Regla de la repetición (REP)**

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

El predicado p es un predicado invariante del while

6. **Regla de consecuencia (CONS)**

$$\frac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

La regla formaliza la posibilidad de reemplazar un predicado p_1 por otro que lo implique, y/o un predicado q_1 por otro al cual implique

- **Llama la atención la forma del axioma de la asignación (ASI).**

  El axioma establece **{p[x|e]} x := e {p}.**

  Se lee así: si luego de *x := e* vale p en términos de x, entonces antes de *x := e* valía p en términos de *e*.

  Por ejemplo, veamos cómo se completa la fórmula {?} x := x + 1 {x ≥ 0}:

  > {x ≥ 0[x|**x + 1**]} x := **x + 1** {x ≥ 0}
  > {**x + 1** ≥ 0} x := **x + 1** {x ≥ 0}
  > Es decir, si luego de *x := x + 1* vale x ≥ 0, entonces antes de *x := x + 1* valía x + 1 ≥ 0.

  El axioma se lee de atrás para adelante (*backward*), lo que impone una forma de probar de la postcondición a la precondición.

  Sería más natural la forma de axioma hacia adelante (*forward*): **{true} x := e {x = e}.**

  Pero esta fórmula de correctitud no siempre es verdadera. Por ejemplo:

  > **{true} x := x + 1 {x = x + 1}** es una fórmula falsa.
  > El contenido de la x de la derecha es previo a la asignación, y el de la x de la izquierda es posterior.

  Existe un axioma correcto hacia adelante, más complicado y no suele usarse: **{p} x := e {∃z: (p[x|z] ∧ x = e[x|z])}.**

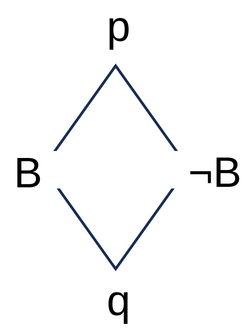
- En la **regla de la secuencia (SEC),** el predicado **r** actúa como **nexo** y luego **se descarta**, no se propaga.

$$\frac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

Se permite usar la siguiente **generalización**:

$$\frac{\{p\}\ S_1\ \{r_1\},\ \{r_1\}\ S_2\ \{r_2\},\ \ldots,\ \{r_{n-1}\}\ S_n\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ ;\ \ldots\ ;\ S_n\ \{q\}}$$

- La **regla del condicional (COND)** formula un modo de verificar una selección condicional fijando un **único punto de entrada** y un **único punto de salida,** correspondientes a **p** y **q**, respectivamente.



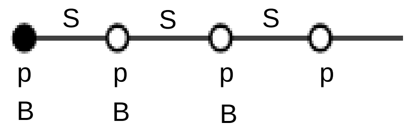
$$\frac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{q\}}$$

- La **regla de la repetición (REP)** se basa en un **invariante p**. Si p vale al comienzo del *while*, y mientras vale B el cuerpo S preserva p, entonces por un razonamiento **inductivo** p vale al finalizar el *while*.

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

Esta inducción se conoce como *inducción computacional.*

Claramente, REP **no asegura que el *while* termine.**



- La **regla de consecuencia (CONS)** permite **reforzar precondiciones** y **debilitar postcondiciones.** P.ej.:

de:	$\{x > 0\}$ S $\{x = 0\}$	de:	$\{true\}$ S $\{x = y + 1\}$
y:	$x > 5 \rightarrow x > 0$	y:	$x = y + 1 \rightarrow x > y$
se deduce:	$\{x > 5\}$ S $\{x = 0\}$	se deduce:	$\{true\}$ S $\{x > y\}$

$$\frac{p \rightarrow p_1,\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

La regla no depende del lenguaje de programación sino del dominio semántico.

Es una regla **semántica** más que sintáctica.

Permite utilizar **todos los axiomas del dominio semántico** (en este caso los enteros), que se agregan a H.

Nota: así, la incompletitud de la aritmética "condena" al método H con los enteros a ser incompleto.
(de todos modos, de lo que se trata es de probar programas, no enunciados de la aritmética)

- El método H es **composicional**:

Dado un programa S con subprogramas $S_1, \ldots, S_n$, que valga la fórmula $\{p\}$ S $\{q\}$ depende sólo de que valgan ciertas fórmulas $\{p_1\}\ S_1\ \{q_1\}, \ldots, \{p_n\}\ S_n\ \{q_n\}$, sin importar el contenido de los S_i (noción de **caja negra**).

P.ej., dado $S :: S_1 ; S_2$,
de $\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$
se deduce $\{p\}\ S_1 ; S_2\ \{q\}$,
independientemente de los contenidos de S_1 y S_2.

Si se tiene $\{r\}\ S_3\ \{q\}$ también se deduce $\{p\}\ S_1 ; S_3\ \{q\}$. S_2 y S_3 son intercambiables por ser funcionalmente equivalentes en relación a (r, q).

Ejemplo 1. Prueba de un programa que intercambia los valores de dos variables

- Dado S_{swap} :: z := x ; x := y ; y := z, se va a probar:

$$\{x = X \wedge y = Y\}\ S_{swap}\ \{y = X \wedge x = Y\}$$

Antes lo probamos **semánticamente**, usando la semántica del lenguaje de programación.

Ahora lo probaremos **sintácticamente**, usando axiomas y reglas (antes hemos mostrado la idea general).

Por la forma del programa S_{swap} recurrimos al axioma ASI tres veces, y al final completamos con la (generalización de la) regla SEC:

1. $\{z = X \wedge x = Y\}$ y := z $\{y = X \wedge x = Y\}$ (ASI)

2. $\{z = X \wedge y = Y\}$ x := y $\{z = X \wedge x = Y\}$ (ASI)

3. $\{x = X \wedge y = Y\}$ z := x $\{z = X \wedge y = Y\}$ (ASI)

4. **$\{x = X \wedge y = Y\}$ z := x ; x := y ; y := z $\{y = X \wedge x = Y\}$** (1, 2, 3, SEC)

Axioma ASI

{p[x|e]} x := e {p}

Regla SEC

$$\frac{\{p\}\ S_1\ \{r\},\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

- Notar cómo el axioma ASI establece una forma de prueba **de la postcondición a la precondición**.

- Por la **sensatez** de H (que vamos a probar la clase que viene), haber probado **sintácticamente**:

$$\{x = X \wedge y = Y\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}$$

significa que la fórmula se cumple **semánticamente.**

- Obviamente, también se cumple **semánticamente** la precondición con los operandos permutados:

$$\{y = Y \wedge x = X\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}$$

- Para probar esta fórmula **sintácticamente** recurrimos a la **regla de consecuencia (CONS):**

Agregamos a la prueba anterior, que terminaba con:

4. $\{x = X \wedge y = Y\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}$ (1, 2, 3, SEC)

los siguientes dos pasos:

5. $(y = Y \wedge x = X) \rightarrow (x = X \wedge y = Y)$ (MAT)

6. $\{y = Y \wedge x = X\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}$ (4, 5, CONS)

- El predicado del paso 5 es un axioma de los enteros, por eso usamos el indicador **MAT** (por matemáticas).

- Así, hemos probado mediante el método H también la fórmula de correctitud:

$$\{y = Y \wedge x = X\}\ S_{swap}\ \{y = X \wedge x = Y\}$$

- Y si quisiéramos **instanciarla**, por ejemplo en:

$$\{y = 2 \wedge x = 1\}\ S_{swap}\ \{y = 1 \wedge x = 2\}$$

debemos recurrir a una nueva regla, la **regla de instanciación (INST):**

$$\frac{f(X)}{f(c)}$$

tal que f es una fórmula de correctitud, X es una variable lógica, y c está en el dominio de X.

INST es una regla universal, se la usa en todos los métodos, a pesar de que no se la suele mencionar.

Ejemplo 2. Prueba de un programa que calcula el valor absoluto

- El siguiente programa calcula en una variable *y* el valor absoluto de una variable x:

$$S_{va} :: \text{ if } x > 0 \text{ then } y := x \text{ else } y := -x \text{ fi}$$

Vamos a probar **{true} S_{va} {y ≥ 0}**

que no es una especificación correcta del programa. Por ejemplo: **S_{va} :: y := 0** no es el programa referido.

Considerando las dos asignaciones del programa, los primeros pasos de la prueba podrían ser:

1. $\{x \geq 0\}\ y := x\ \{y \geq 0\}$ (ASI)

2. $\{-x \geq 0\}\ y := -x\ \{y \geq 0\}$ (ASI)

- Por el if then else, hay que usar la regla COND.

- Necesitamos fórmulas $\{p \wedge x > 0\}\ y := x\ \{y \geq 0\}$ y $\{p \wedge \neg(x > 0)\}\ y := -x\ \{y \geq 0\}$

- Usaremos **p = true**:

- Quedará $\{true \wedge x > 0\}\ y := x\ \{y \geq 0\}$ y $\{true \wedge \neg(x > 0)\}\ y := -x\ \{y \geq 0\}$

- Y faltará reemplazar $true \wedge x > 0$ con $x \geq 0$, y $true \wedge \neg(x > 0)$ con $-x \geq 0$

Axioma ASI

$\{p[x|e]\}\ x := e\ \{p\}$

Regla COND

$$\frac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{q\}}$$

Pasando en limpio:

1. $\{x \geq 0\}\ y := x\ \{y \geq 0\}$ (ASI)
2. $\{-x \geq 0\}\ y := -x\ \{y \geq 0\}$ (ASI)

Luego hacemos:

3. $(\text{true} \wedge x > 0) \rightarrow x \geq 0$ (MAT)
4. $(\text{true} \wedge \neg(x > 0)) \rightarrow -x \geq 0$ (MAT)

Regla COND

$$\frac{\{p \wedge B\}\ S_1\ \{q\},\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{q\}}$$

y completamos con:

5. $\{\text{true} \wedge x > 0\}\ y := x\ \{y \geq 0\}$ (1, 3, CONS)
6. $\{\text{true} \wedge \neg(x > 0)\}\ y := -x\ \{y \geq 0\}$ (2, 4, CONS)
7. **$\{\text{true}\}$ if $x > 0$ then $y := x$ else $y := -x$ fi $\{y \geq 0\}$** (5, 6, COND)

Una especificación correcta de un programa de valor absoluto es: **$(x = X,\ y = |X|)$**

Ejercicio: verificar el programa anterior S_{va} con respecto a esta especificación.

Ejemplo 3. Prueba de correctitud parcial de un programa que calcula el factorial

- Se verá en la clase práctica. La idea es probar con el método H:

$$\{x > 0\}$$
$$S_{fac} :: a := 1 \,;\, y := 1 \,;\, \text{while } a < x \text{ do } a := a + 1 \,;\, y := y \,.\, a \text{ od}$$
$$\{y = x!\}$$

Es decir que dado $x > 0$, el programa S_{fac}, **si termina**, obtiene $y = x!$

La especificación ($x > 0$, $y = x!$) para un programa que calcule el factorial no es correcta.

Regla REP

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

- Se propone como **invariante** del while:

$$\mathbf{p = (y = a! \wedge a \leq x)}$$

La idea es que se vaya calculando en la variable *y* el factorial de *a*, hasta que se alcance $a = x$. Notar que el invariante propuesto es similar a la postcondición, **la generaliza**, sustituyendo x por *a*. Cuando $a = x$, entonces $y = x!$.

- La prueba se puede estructurar de la siguiente manera:

a) $\{x > 0\}\ a := 1 \,;\, y := 1\ \{y = a! \wedge a \leq x\}$ — Se cumple el invariante por primera vez

b) $\{y = a! \wedge a \leq x\}$
$\quad \text{while } a < x \text{ do } a := a + 1 \,;\, y := y \,.\, a \text{ od}$ — Aplicación de REP y CONS
$\quad \{y = x!\}$

c) $\{x > 0\}\ S_{fac}\ \{y = x!\}$ — Aplicación de SEC sobre (a) y (b)

3.2. Axiomática para la correctitud total (Método H*)

- El método H* es el método H pero con la regla REP del while ampliada.
- La regla REP sólo permite probar la **invariancia de un predicado p**.
- En el método H* se usa **la regla REP***, que permite probar además la **terminación del while**.
- La regla REP* se basa en un **predicado invariante p** y una **función variante t.**

  t es una función entera definida, como el invariante, en términos de las variables de programa.

Regla REP*:

$$\frac{\langle p \wedge B\rangle\ S\ \langle p\rangle,\quad \langle p \wedge B \wedge (t = Z)\rangle\ S\ \langle t < Z\rangle,\quad p \rightarrow t \geq 0}{\langle p\rangle\ \text{while } B \text{ do } S \text{ od } \langle p \wedge \neg B\rangle}$$

while B do t_1

S t_2

od .... t_k

La función t representa el máximo número de iteraciones del while

Se agregan dos premisas a REP, y se reemplazan los símbolos { } por ⟨ ⟩.

Por la segunda premisa, **t se decrementa en cada iteración.**
Z es una variable lógica, no aparece en p, B, t ni S.
El objetivo de Z es fijar el valor de t antes de la ejecución de S.

Por la tercera premisa, **t se mantiene ≥ 0 a lo largo de todo el while.**

($\mathcal{N}$, <) es un ejemplo de relación de orden bien fundada (todo subconjunto tiene minimal).

Así, el while debe terminar, porque en $\mathcal{N}$ (números naturales) **no hay cadenas descendentes infinitas**

Ejemplo 4. Prueba de terminación del programa del factorial

Regla REP*

$$\frac{\langle p \wedge B\rangle\ S\ \langle p\rangle\ ,\ \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle\ ,\ p \rightarrow t \geq 0}{\langle p\rangle \text{ while } B \text{ do } S \text{ od } \langle p \wedge \neg B\rangle}$$

- Se verá en la clase práctica.

- Con el método H se prueba:

  $\{x > 0\}\ S_{fac} :: a := 1 ; y := 1 ; \text{while } a < x \text{ do } a := a + 1 ; y := y . a \text{ od } \{y = x!\}$

  utilizando p.ej. el invariante: **$p = (y = a! \wedge a \leq x)$**

- Con el método H* se puede probar:

  $\langle x > 0\rangle\ S_{fac} :: a :=1 ; y := 1 ; \text{while } a < x \text{ do } a := a + 1 ; y := y . a \text{ od } \langle true\rangle$

  y de esta manera, se puede probar:

  $\langle x > 0\rangle\ S_{fac} :: a :=1 ; y := 1 ; \text{while } a < x \text{ do } a := a + 1 ; y := y . a \text{ od } \langle y = x!\rangle$

- Se propone:

  Invariante **$p = (a \leq x)$**

  Al comienzo del while, $a \leq x$, y a la salida, $a = x$.
  Notar que este invariante es más simple que el utilizado antes.

  Variante **$t = x – a$**

  Al comienzo del while, $t \geq 0$.
  A lo largo de sus iteraciones, t se decrementa en 1.
  Y al final se cumple $t = 0$.

- **¿Por qué separar la prueba del while en dos, correctitud parcial y terminación?**

- Porque son dos pruebas distintas, utilizan técnicas distintas:
  - Correctitud parcial: **inducción** (invariante que vale al comienzo y luego de cada iteración).
  - Terminación: **relación de orden bien fundada** (variante entero ≥ 0 que se decrementa tras cada iteración).

- En verdad, la regla REP* permite probar todo junto, pero la recomendación es probar $\langle p \rangle$ S $\langle q \rangle$ en dos pasos:

(a) $\{p\}$ S $\{q\}$
(b) $\langle p \rangle$ S $\langle true \rangle$

(a) **Si S termina** a partir de p, lo hace en q.
(b) **S termina** a partir de p.
(a, b) **S termina a partir de p en q.**

$$\frac{\langle p \wedge B \rangle\ S\ \langle p \rangle, \quad \langle p \wedge B \wedge (t = Z) \rangle\ S\ \langle t < Z \rangle, \quad p \rightarrow t \geq 0}{\langle p \rangle\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \langle q \rangle}$$
, tal que $(p \wedge \neg B) \rightarrow q$

- Como en el último ejemplo:

$$\langle x > 0 \rangle\ S_{fac} :: a := 1 ;\ y := 1 ;\ \text{while } a < x \text{ do } a := a + 1 ;\ y := y \,.\, a \text{ od } \langle y = x! \rangle$$

Para probar $\{x > 0\}$ S_{fac} $\{y = x!\}$ se utiliza el invariante $\mathbf{y = a! \wedge a \leq x}$,

Y para probar $\langle x > 0 \rangle$ S_{fac} $\langle true \rangle$, un invariante más simple: $\mathbf{a \leq x}$. **evitando arrastrar información innecesaria.**

Anexo

Proof outlines (esquemas de prueba)

- Presentación alternativa de una prueba: se intercalan los pasos de la prueba entre las instrucciones del programa.

- Se obtiene una prueba más estructurada, que documenta adecuadamente el programa.

- En la verificación de los programas concurrentes las *proof outlines* son **imprescindibles**.

- Por ejemplo, la siguiente es una *proof outline* de correctitud parcial de un programa que calcula el factorial:

```
{x > 0}
Sfac :: a := 1 ; y := 1 ;
        while a < x do
               a := a + 1 ; y := y . a
        od
{y = x!}
```

{x > 0}
a := 1 ;
y := 1 ;
{inv: y = a! ∧ a ≤ x}
while a < x do
{y = a! ∧ a < x}
a := a + 1 ;
y := y . a
od
{(y = a! ∧ a ≤ x) ∧ ¬(a < x)}
{y = x!}

En una proof outline de correctitud total se agrega el variante.

- Mínimamente se suelen documentar la **precondición**, la **postcondición,** los **invariantes** y los **variantes**.

Invariantes, variantes, y propiedades *safety* (seguridad) y *liveness* (progreso)

- Todo predicado utilizado en una prueba es en realidad un **invariante**. Volviendo a las *proof outlines*: cualquiera sea el estado inicial, todo predicado siempre se cumple en el lugar donde se establece:

{x > 0}
 a := 1 ; y := 1 ;
{y = a! ∧ a ≤ x}
 while a < x do
{y = a! ∧ a < x}
 a := a + 1 ; y := y . a
 od
{y = x!}

- El uso de un invariante para la prueba de un *while* determina una prueba por **inducción**: es un predicado que vale al comienzo del *while* (base inductiva) y que toda iteración preserva (paso inductivo). De esta manera se prueba que el invariante se cumple **a lo largo de toda la computación** del *while*.

- Ligado a lo anterior, la correctitud parcial pertenece a la familia de las propiedades ***safety***. Son propiedades que se prueban por **inducción**. Se asocian al enunciado: “Algo malo no puede suceder”. Otros ejemplos son la *ausencia de deadlock* y la *exclusión mutua* o *ausencia de interferencia*, en los programas concurrentes.

- La terminación o no divergencia, en cambio, pertenece a la familia de las propiedades **liveness**, basadas en **funciones variantes** definidas en **relaciones de orden bien fundadas.** Se asocian al enunciado: “Algo bueno va a suceder”. Otro ejemplo es la *ausencia de inanición (non starvation),* en los programas concurrentes.

Axiomas y reglas adicionales

- Por la **completitud** del método de prueba estudiado (que se prueba en la clase que viene), agregarle axiomas y reglas resulta **redundante**. De todos modos, esta práctica es usual en los sistemas deductivos, facilita y acorta las pruebas. Algunos ejemplos clásicos de axiomas y reglas adicionales son:

- Axioma de invariancia (INV) $\{p\}\ S\ \{p\}$

  Cuando las variables libres de p y las variables que modifica S **son disjuntas**.

- Regla de la disyunción (OR)

$$\frac{\{p\}\ S\ \{q\},\ \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$$

  Util para una **verificación por casos**.

- Regla de la conjunción (AND)

$$\frac{\{p_1\}\ S\ \{q_1\},\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

  También útil para una **verificación por casos**.

- El axioma INV suele emplearse en combinación con la regla AND, para producir la **Regla de invariancia (RINV):**

$$\frac{\{p\}\ S\ \{q\}}{\{r \wedge p\}\ S\ \{r \wedge q\}}$$

tal que **ninguna variable libre de r es modificable por S** (se cumple {r} S {r}).

- Dada la Regla de la Disyunción (OR):

$$\frac{\{p\}\ S\ \{q\}\ ,\ \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$$

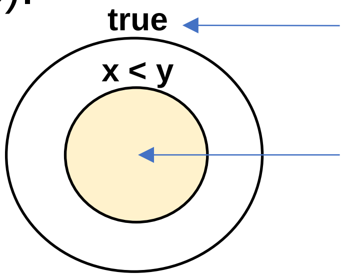
una forma particular de la regla de uso habitual es:

$$\frac{\{p \wedge s\}\ S\ \{q\}\ ,\ \{p \wedge \neg s\}\ S\ \{q\}}{\{p\}\ S\ \{q\}}$$

útil cuando la prueba de {p} S {q} se facilita reforzando la precondición con **dos aserciones complementarias**.

Sintaxis y semántica. Abstracción.

- Un **predicado** (elemento sintáctico) representa un **conjunto de estados** (elemento semántico), el conjunto de todos los estados que satisfacen el predicado. P.ej., $x < y$ denota a todos los estados tales que $x < y$. En particular, **true** denota a todos los estados y **false** denota al conjunto vacío de estados (para todo estado σ, se cumple $\sigma \models$ true y $\sigma \not\models$ false).



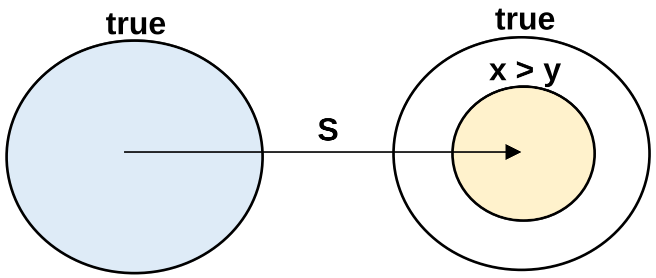
- Una **especificación** de un programa S es un par de predicados (p, q) asociados a la **entrada** y la **salida** de S, respectivamente, p denota al conjunto de estados iniciales de S, y q al conjunto de estados finales de S.

- Por ejemplo, ($x = X$, $x = 2X$) es satisfecha por un programa que duplica su entrada x (como S :: $x := x + x$). La variable x es una **variable de programa.** La variable X es una **variable lógica o de especificación** (no es parte del programa, se usa para fijar valores).

Ejercicio. Especificar un programa S que termine con la postcondición $x > y$.

Una posible especificación podría ser: ($x = X \wedge y = Y$, $x > y$)

Otra más simple podría ser: (true, $x > y$)



A partir de la precondición true, el programa S debe terminar en la postcondición x > y

Lema de Separación

Prueba formal de que $<p>\ S\ <q> \leftrightarrow (\{p\}\ S\ \{q\} \wedge\ <p>\ S\ <\mathrm{true}>)$

- Primero probamos $<p>\ S\ <q> \rightarrow (\{p\}\ S\ \{q\} \wedge\ <p>\ S\ <\mathrm{true}>)$:

  Sea $<p>\ S\ <q>$. Entonces, dado σ, vale $\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \wedge val(\pi(S, \sigma)) \models q)$. Por lo tanto:
  (a) $(\sigma \models p \wedge val(\pi(S, \sigma)) \neq \perp) \rightarrow val(\pi(S, \sigma)) \models q$, es decir $\{p\}\ S\ \{q\}$.
  (b) $\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \wedge val(\pi(S, \sigma)) \models \mathrm{true})$, es decir $<p>\ S\ <\mathrm{true}>$.

  Así, por (a) y (b): $\{p\}\ S\ \{q\} \wedge\ <p>\ S\ <\mathrm{true}>$.

- Ahora probamos $(\{p\}\ S\ \{q\} \wedge\ <p>\ S\ <\mathrm{true}>) \rightarrow\ \models\ <p>\ S\ <q>$:

  Sea $\{p\}\ S\ \{q\} \wedge\ <p>\ S\ <\mathrm{true}>$. Entonces, dado σ, vale:
  (a) $(\sigma \models p \wedge val(\pi(S, \sigma)) \neq \perp) \rightarrow val(\pi(S, \sigma)) \models q$.
  (b) $\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \wedge val(\pi(S, \sigma)) \models \mathrm{true})$.
  (c) Por (b): $\sigma \models p \rightarrow val(\pi(S, \sigma)) \neq \perp$.
  (d) Por (a) y (b): $\sigma \models p \rightarrow val(\pi(S, \sigma)) \models q$.

  Así, por (c) y (d): $\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \wedge val(\pi(S, \sigma)) \models q)$, es decir $<p>\ S\ <q>$.

Sensatez total

- Se probará en la práctica: $\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r < y \wedge r \geq 0\}$

  siendo: S_{div} :: c := 0 ; r := x ; while r ≥ y do r := r – y ; c := c + 1 od

  y la estructura general de la prueba (p es el invariante):

  a) $\{x \geq 0 \wedge y > 0\}$ c := 0 ; r := x {p}
  b) {p} while r ≥ y do r := r – y ; c := c + 1 od $\{p \wedge \neg(r \geq y)\}$
  c) $\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y \,.\, c + r \wedge r < y \wedge r \geq 0\}$

La misma prueba sirve para probar el programa con variables reales.

Completitud aritmética

- Sea el siguiente programa, con variables x y Ɛ de tipo real inicialmente mayores que 0:

```
S :: while x > Ɛ
        do
        x := x / 2
        od
```

La función t tiene que ser entera, independientemente del tipo de las variables de programa.

  Se cumple: $\langle x = X \wedge X > 0 \wedge \mathcal{E} > 0\rangle\ S\ \langle true\rangle$

  y una posible función entera t para la prueba de terminación es: t = if x > Ɛ then $[\log_2 X/\mathcal{E}] - \log_2 X/x$ else 0 fi

Clase práctica 11

Ejemplo 1. Prueba de correctitud parcial de un programa que calcula el factorial

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

Recordatorio de la regla REP

- Vamos a probar con el método H:

$$\{x > 0\}$$
$$S_{fac} :: a := 1 \ ; \ y := 1 \ ; \ \text{while } a < x \text{ do } a := a + 1 \ ; \ y := y \, . \, a \text{ od}$$
$$\{y = x!\}$$

Es decir, dado $x > 0$, el programa S_{fac}, si termina, obtiene $y = x!$

Ejercicio: ¿Es correcta la especificación ($x > 0$, $y = x!$) para un programa que calcule el factorial?

- Se propone como **invariante** del while:

$$\mathbf{p = (y = a! \wedge a \leq x)}$$

La idea es que se vaya calculando en *y* el factorial de *a*, hasta que $a = x$. En efecto, notar que el invariante propuesto es similar a la postcondición, **la generaliza**, sustituyendo x por *a*. Cuando $a = x$, entonces $y = x!$.

- La prueba se puede estructurar de la siguiente manera:

a) $\{x > 0\}\ a := 1 \ ; \ y := 1\ \{y = a! \wedge a \leq x\}$ — Se cumple el invariante por primera vez

b) $\{y = a! \wedge a \leq x\}$
while $a < x$ do $a := a + 1 \ ; \ y := y \, . \, a$ od
$\{y = x!\}$ — Aplicación de REP y CONS

c) $\{x > 0\}\ S_{fac}\ \{y = x!\}$ — Aplicación de SEC sobre (a) y (b)

Prueba de (a)

1. $\{1 = a! \wedge a \leq x\}\ y := 1\ \{y = a! \wedge a \leq x\}$ (ASI)
2. $\{1 = 1! \wedge 1 \leq x\}\ a := 1\ \{1 = a! \wedge a \leq x\}$ (ASI)
3. $\{x > 0\}\ a := 1;\ y := 1\ \{y = a! \wedge a \leq x\}$ (1, 2, SEC, CONS)

Hemos abreviado pasos. En el paso 3 se usa CONS considerando $(x > 0) \longrightarrow (1 = 1! \wedge 1 \leq x)$.

Prueba de (b)

4. $\{y\ .\ a = a! \wedge a \leq x\}\ y := y\ .\ a\ \{y = a! \wedge a \leq x\}$ (ASI)
5. $\{y\ .\ (a + 1) = (a + 1)! \wedge (a + 1) \leq x\}\ a := a + 1\ \{y\ .\ a = a! \wedge a \leq x\}$ (ASI)
6. $\{y = a! \wedge a \leq x \wedge \mathbf{a < x}\}\ a := a + 1;\ y := y\ .\ a\ \{y = a! \wedge a \leq x\}$ (4, 5, SEC, CONS)
7. $\{y = a! \wedge a \leq x\}$ while $\mathbf{a < x}$ do $a := a + 1;\ y := y\ .\ a$ od $\{y = a! \wedge a \leq x \wedge \mathbf{\neg(a < x)}\}$ (6, REP)
8. $\{y = a! \wedge a \leq x\}$ while $a < x$ do $a := a + 1;\ y := y\ .\ a$ od $\{y = x!\}$ (7, CONS)

En el paso 6 se usa CONS considerando $(y = a! \wedge a \leq x \wedge a < x) \longrightarrow (y\ .\ (a + 1) = (a + 1)! \wedge (a + 1) \leq x)$.

En el paso 8 se usa CONS considerando $(y = a! \wedge a \leq x \wedge \neg(a < x)) \longrightarrow (y = x!)$.

Prueba de (c)

9. $\{x > 0\}\ S_{fac}\ \{y = x!\}$ (3, 8, SEC)

Ejemplo 2. Prueba de terminación del programa del factorial

$$\frac{\langle p \wedge B\rangle\ S\ \langle p\rangle\ ,\ \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle\ ,\ p \rightarrow t \geq 0}{\langle p\rangle\ \text{while } B \text{ do } S \text{ od } \langle p \wedge \neg B\rangle}$$

Recordatorio de la regla REP*

- Con el método H hemos probado:

  $\{x > 0\}\ S_{fac} :: a := 1\ ;\ y := 1\ ;\ \text{while } a < x \text{ do } a := a + 1\ ;\ y := y\ .\ a \text{ od } \{y = x!\}$

  utilizando el invariante $\mathbf{p = (y = a! \wedge a \leq x)}$

- Con el método H* probaremos:

  $\langle x > 0\rangle\ S_{fac} :: a := 1\ ;\ y := 1\ ;\ \text{while } a < x \text{ do } a := a + 1\ ;\ y := y\ .\ a \text{ od } \langle true\rangle$

  y de esta manera habremos probado:

  $\langle x > 0\rangle\ S_{fac} :: a := 1\ ;\ y := 1\ ;\ \text{while } a < x \text{ do } a := a + 1\ ;\ y := y\ .\ a \text{ od } \langle y = x!\rangle$

- Se propone:

Invariante $\mathbf{p = (a \leq x)}$	Al comienzo del while, $a \leq x$, y a la salida, $a = x$. **Notar que este invariante es más simple que el utilizado antes.**
Variante $\mathbf{t = x - a}$	Al comienzo del while, $t \geq 0$. A lo largo de sus iteraciones, t se decrementa en 1. Y al final se cumple $t = 0$.

Inicializaciones:
1. $\langle x > 0 \rangle$ a := 1 ; y := 1 $\langle a \leq x \rangle$ (ASI, SEC, CONS)

Repetición:
Se tiene: $p = (a \leq x)$, $t = x - a$
Hay que probar por REP* tres premisas:

Premisa 1: $\langle p \wedge B \rangle\ S\ \langle p \rangle$
2. $\langle a \leq x \wedge a < x \rangle$ a := a + 1 ; y := y . a $\langle a \leq x \rangle$ (ASI, SEC, CONS)

Premisa 2: $\langle p \wedge B \wedge t = Z \rangle\ S\ \langle t < Z \rangle$
3. $\langle a \leq x \wedge a < x \wedge x - a = Z \rangle$ a := a + 1 ; y := y . a $\langle x - a < Z \rangle$ (ASI, SEC, CONS)

Premisa 3: $p \rightarrow t \geq 0$
4. $a \leq x \rightarrow x - a \geq 0$ (MAT)

Conclusión: $\langle p \rangle$ while B do S od $\langle p \wedge \neg B \rangle$
5. $\langle a \leq x \rangle$ while a < x do a := a + 1 ; y := y . a od $\langle a \leq x \wedge \neg(a < x) \rangle$ (2, 3, 4, REP*)

Programa completo:
6. $\langle x > 0 \rangle$ a :=1 ; y := 1 ; while a < x do a := a + 1 ; y := y . a od $\langle true \rangle$ (1, 5, SEC, CONS)

En realidad, se probó $\langle x > 0 \rangle\ S_{fac}\ \langle a = x \rangle$. Pero la postcondición en este caso es irrelevante, el objetivo fue asegurar la terminación del while. La postcondición y = x! se alcanzó en la prueba de correctitud parcial.

$$\frac{\langle p \wedge B \rangle\ S\ \langle p \rangle\ ,\ \langle p \wedge B \wedge t = Z \rangle\ S\ \langle t < Z \rangle\ ,\ p \rightarrow t \geq 0}{\langle p \rangle\ \text{while } B \text{ do } S \text{ od}\ \langle p \wedge \neg B \rangle}$$

Recordatorio de la regla REP*

Programa

```
a :=1 ;
y := 1 ;
while a < x
do
a := a + 1 ;
y := y . a
od
```

Proof outline

```
⟨x > 0⟩
a :=1 ; y := 1 ;
⟨inv: a ≤ x, var: x - a⟩
while a < x
do
a := a + 1 ; y := y . a
od
⟨a ≤ x ∧ ¬(a < x)⟩
⟨true⟩
```

Ejemplo 3. Prueba de correctitud parcial de un programa de división entera

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

Recordatorio de la regla REP

Probaremos con H: $\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y\,.\,c + r \wedge r < y \wedge r \geq 0\}$, con:

S_{div} :: $c := 0$; $r := x$; while $r \geq y$ do $r := r - y$; $c := c + 1$ od

Proponemos como invariante del *while* la aserción:

$$p = (x = y\,.\,c + r \wedge r \geq 0)$$

obtenida por una **generalización** de la postcondición del *while*.

Notar que cuando el programa termina se cumple $r < y$, y así de la conjunción de esta condición y el invariante se alcanza la postcondición buscada.

Podemos estructurar la prueba de la siguiente manera, que se desarrolla en el slide siguiente:

a) $\{x \geq 0 \wedge y > 0\}\ c := 0 ; r := x\ \{p\}$

b) $\{p\}$ while $r \geq y$ do $r := r - y$; $c := c + 1$ od $\{p \wedge \neg(r \geq y)\}$

c) Finalmente, aplicando SEC a (a) y (b), y como $(p \wedge \neg(r \geq y)) \rightarrow x = y\,.\,c + r \wedge r < y \wedge r \geq 0$, por CONS se llega a:

$$\mathbf{\{x \geq 0 \wedge y > 0\}\ S_{div}\ \{x = y\,.\,c + r \wedge r < y \wedge r \geq 0\}}$$

Dada la precondición $\{x \geq 0 \wedge y > 0\}$ y la postcondición $\{x = y . c + r \wedge r < y \wedge r \geq 0\}$, usamos el invariante $\{x = y . c + r \wedge r \geq 0\}$:

Prueba de (a)

1. $\{x = y . c + x \wedge x \geq 0\}$ r := x $\{x = y . c + r \wedge r \geq 0\}$ (ASI)
2. $\{x = y . 0 + x \wedge x \geq 0\}$ c := 0 $\{x = y . c + x \wedge x \geq 0\}$ (ASI)
3. $\{x = y . 0 + x \wedge x \geq 0\}$ c := 0 ; r := x $\{x = y . c + r \wedge r \geq 0\}$ (1, 2, SEC)
4. $(x \geq 0 \wedge y > 0) \rightarrow (x = y . 0 + x \wedge x \geq 0)$ (MAT)
5. $\{x \geq 0 \wedge y > 0\}$ c := 0 ; r := x $\{x = y . c + r \wedge r \geq 0\}$ (3, 4, CONS)

Prueba de (b)

6. $\{x = y . (c + 1) + r \wedge r \geq 0\}$ c := c + 1 $\{x = y . c + r \wedge r \geq 0\}$ (ASI)
7. $\{x = y . (c + 1) + (r - y) \wedge r - y \geq 0\}$ r := r – y $\{x = y . (c + 1) + r \wedge r \geq 0\}$ (ASI)
8. $\{x = y . (c + 1) + (r - y) \wedge r - y \geq 0\}$ r := r – y ; c := c + 1 $\{x = y . c + r \wedge r \geq 0\}$ (6, 7, SEC)
9. $(x = y . c + r \wedge r \geq 0 \wedge r \geq y) \rightarrow (x = y . (c + 1) + (r - y) \wedge r - y \geq 0)$ (MAT)
10. $\{x = y . c + r \wedge r \geq 0 \wedge r \geq y\}$ r := r – y ; c := c + 1 $\{x = y . c + r \wedge r \geq 0\}$ (8, 9, CONS)
11. $\{x = y . c + r \wedge r \geq 0\}$ while $r \geq y$ do r := r – y ; c := c + 1 od $\{x = y . c + r \wedge r \geq 0 \wedge \neg(r \geq y)\}$ (10, REP)

Prueba de (c)

12. $\{x \geq 0 \wedge y > 0\}$ c := 0 ; r := x ; while $r \geq y$ do r := r – y ; c := c + 1 od $\{ x = y . c + r \wedge r \geq 0 \wedge \neg(r \geq y)\}$ (5, 11, SEC)
13. $(x = y . c + r \wedge r \geq 0 \wedge \neg(r \geq y)) \rightarrow (x = y . c + r \wedge r < y \wedge r \geq 0)$ (MAT)
14. $\{x \geq 0 \wedge y > 0\}$ c := 0 ; r := x ; while $r \geq y$ do r := r – y ; c := c + 1 od $\{x = y . c + r \wedge r < y \wedge r \geq 0\}$ (12, 13, CONS)

También se puede probar $\{x \geq 0 \wedge y \geq 0\}$ S_{div} $\{x = y . c + r \wedge r < y \wedge r \geq 0\}$, **permitiendo que el divisor *y* sea 0**. La prueba queda como ejercicio.

<u>**Ejemplo 4. Prueba de terminación del programa de división entera**</u>

$$\frac{\langle p \wedge B \rangle\ S\ \langle p \rangle\ ,\ \langle p \wedge B \wedge t = Z \rangle\ S\ \langle t < Z \rangle\ ,\ p \rightarrow t \geq 0}{\langle p \rangle \text{ while } B \text{ do } S \text{ od } \langle p \wedge \neg B \rangle}$$

Recordatorio de la regla REP*

Ya hemos probado mediante el método H:

$\{x \geq 0 \wedge y > 0\}$

S_{div} :: c := 0 ; r := x ; while r ≥ y do r := r – y ; c := c + 1 od

$\{x = c.y + r \wedge 0 \leq r < y\}$

Notamos que también se puede probar con $y \geq 0$.

Ahora probaremos la terminación de S_{div} empleando REP*. Naturalmente acá necesariamente la precondición debe incluir $y > 0$.
Vamos a verificar:

$$\langle x \geq 0 \wedge y > 0 \rangle\ S_{div}\ \langle true \rangle$$

Se propone como **invariante** del while la aserción:

$$p = (x = c\ .\ y + r \wedge r \geq 0 \wedge y > 0)$$

En este caso el invariante no es más simple que el de la prueba de correctitud parcial.

Y se propone como **variante** del while la función:

$$t = r$$

Claramente t siempre es positiva y se decrementa en cada iteración.

La prueba es la siguiente (para simplificar la escritura usamos directamente p en lugar de $x = c.y + r \wedge r \geq 0 \wedge y > 0$):

Prueba del fragmento inicial del programa:

1. $\langle x \geq 0 \wedge y > 0 \rangle$ c := 0 ; r := x $\langle p \rangle$ (ASI, SEC, CONS)

Prueba de las tres premisas de REP*:

2. $\langle p \wedge r \geq y \rangle$ r := r – y ; c := c + 1 $\langle p \rangle$ (ASI, SEC, CONS)

3. $\langle p \wedge r \geq y \wedge r = Z \rangle$ r := r – y ; c := c + 1 $\langle r < Z \rangle$ (ASI, SEC, CONS)

4. $p \rightarrow r \geq 0$ (MAT)

Se establece la conclusión de REP* y la fórmula final pretendida:

5. $\langle p \rangle$ while r ≥ y do r := r – y ; c := c + 1 od $\langle p \wedge \neg(r \geq y) \rangle$ (2, 3, 4, REP*)

6. $\langle x \geq 0 \wedge y > 0 \rangle$ S_{div} $\langle true \rangle$ (1, 5, SEC, CONS)