Practica 4 – Clases 10 – 11

Verificación de programas

Hay: un lenguaje de especificación para describir los problemas (especificaciones) y un lenguaje de programación para describir las soluciones (programas).

Planteamos una metodología para verificar la correctitud de un programa con respecto a una especificación.

Programas imperativos: Transforman estados con instrucciones. Secuenciales. De entrada/salida.

- Approach natural para verificar programas: Operacional. Se analizan semánticamente los programas (prohibitivo cuando se tornan complejos)

- Approach a estudiar: Axiomático, basado en la lógica de predicados, con axiomas y reglas de inferencia asociados a las instrucciones de los programas. Pruebas sintácticas.

En la prueba es sintáctica, sólo se usan axiomas, reglas y teoremas demostrados previamente. Se exige que una axiomática tiene que ser sensata (lo que pruebe se cumpla semánticamente). Y es deseable que sea completa (pueda probar todo lo que se cumpla semánticamente, esto no siempre se cumple)

Idea central → Construir y verificar programas en simultáneo.

Un programa es correcto siempre con respecto a una especificación.

Lenguaje de programación PLW

Sintaxis en BNF

Sus instrucciones son:

$$S :: \text{skip} \mid x := e \mid S_1 \,;\, S_2 \mid \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{while } B \text{ do } S_1 \text{ od}$$

La expresión e es de tipo entero:

$$e :: n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \,.\, e_2 \mid \ldots \mid \text{if } B \text{ then } e_1 \text{ else } e_2 \text{ fi}$$

La expresión B es de tipo booleano:

$$B :: \text{true} \mid \text{false} \mid e_1 = e_2 \mid e_1 < e_2 \mid \ldots \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2 \mid \ldots$$

Por ejemplo, el siguiente programa calcula en la variable y el factorial de un número entero $x > 0$:

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

Semántica (matemática o denotacional) de las expresiones

A cada constante se le asigna un valor fijo entero o booleano. Algunos casos:

1. A la constante n, el número n.

2. A las constantes true y false, los valores verdadero y falso.

3. Al símbolo +, la función suma.

4. Al símbolo ¬, la función negación.

5. Al símbolo =, la función igualdad.

A cada variable se le asigna un valor no fijo con una función σ llamada estado. La expresión $\sigma(x)$ denota el contenido de la variable x según el estado σ, y el conjunto de todos los estados se denota con Σ.

A cada expresión se le asigna inductivamente, mediante una función S, un valor (número o valor de verdad) que depende del estado considerado:

1. $S(n)(\sigma) = n$.
2. $S(true)(\sigma) =$ verdadero.
3. $S(x)(\sigma) = \sigma(x)$.
4. $S(e1 + e2)(\sigma) = S(e1)(\sigma) + S(e2)(\sigma)$.
5. $S(e1 = e2)(\sigma) = (S(e1)(\sigma) = S(e2)(\sigma))$.
6. $S(\neg B)(\sigma) = \neg S(B)(\sigma)$

Se usa $\sigma(e)$ para abreviar $S(e)(\sigma)$ y $\sigma(B)$ para abreviar $S(B)(\sigma)$. Para denotar que una variable x tiene un valor particular n en un estado σ, se utiliza la expresión $\sigma[x \mid n]$

Semántica (operacional) de las instrucciones

Se define mediante una relación → de transición entre configuraciones, que son pares (S, σ), siendo S una continuación sintáctica y σ un estado:

1. $(skip, \sigma) \rightarrow (E, \sigma)$

   El skip se consume en un paso (es atómico) y no modifica el estado inicial. E es la continuación sintáctica vacía.

2. $(x := e, \sigma) \rightarrow (E, \sigma[x|\sigma(e)])$

   La asignación también es atómica y el estado final es como el inicial salvo que ahora la variable x tiene el valor de la expresión e.

3. Si $(S, \sigma) \rightarrow (S', \sigma')$, entonces $(S ; T, \sigma) \rightarrow (S' ; T, \sigma')$ para toda instrucción T

   La secuencia se ejecuta de izquierda a derecha. Una vez consumido S, si no diverge, se ejecuta T. Se define $E ; S = S ; E = S$.

4. Si $\sigma(B) =$ verdadero, entonces $(\text{if } B \text{ then } S1 \text{ else } S2 \text{ fi}, \sigma) \rightarrow (S1, \sigma)$

   $=$ falso, entonces $(\text{if } B \text{ then } S1 \text{ else } S2 \text{ fi}, \sigma) \rightarrow (S2, \sigma)$

La evaluación de la expresión B es atómica y no modifica el estado inicial. Su evaluación determina si se ejecuta S1 o S2 .

5. Si $\sigma(B)$ = verdadero, entonces (while B do S od, σ) → (S ; while B do S od, σ)

= falso, entonces (while B do S od, σ) → (E, σ)

La evaluación de B es atómica y no modifica el estado inicial. Su evaluación determina si se ejecuta S o se termina la instrucción.

Dados un programa S y un estado σ, a dicha configuración inicial (S, σ) se le asocia una computación $\pi(S, \sigma)$, que es la secuencia de configuraciones producida por la ejecución de S a partir de σ. $val(\pi(S, \sigma))$ denota el estado final de $\pi(S, \sigma)$. Si $\pi(S, \sigma)$ es infinita, se usa $val(\pi(S, \sigma)) = \perp$.

Lenguaje de especificación

<u>Sintaxis (en BNF)</u>

Es la del lenguaje de la lógica de predicados. En este contexto los predicados se conocen como aserciones. Las aserciones tienen la forma:

$$p :: true \mid false \mid e1 = e2 \mid e1 < e2 \mid \ldots \mid \neg p \mid p1 \vee p2 \mid \ldots \mid \exists x{:}p \mid \forall x{:}p$$

Es decir, toda expresión booleana es una aserción. Además, las aserciones pueden tener cuantificadores.

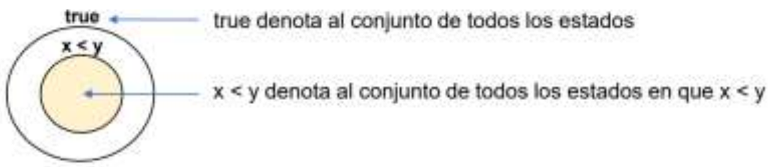
Semántica (matemática o denotacional)

A cada aserción se le asigna inductivamente, mediante la función S definida antes, un valor de verdad que depende del estado considerado.

1. $S(true)(\sigma)$ = verdadero
2. $S(\neg p)(\sigma) = \neg S(p)(\sigma)$
3. $S(p \vee q)(\sigma) = S(p)(\sigma) \vee S(q)(\sigma)$
4. $S(\exists x{:}\ p)(\sigma)$ = verdadero sii $S(p)(\sigma[x|n])$ = verdadero para algún número n
5. $S(\forall x{:}\ p)(\sigma)$ = verdadero sii $S(p)(\sigma[x|n])$ = verdadero para todo número n

Si S(p)(σ) = verdadero, se dice que σ satisface p, o que p es verdadera cuando se evalúa en σ. La notación σ |= p abrevia S(p)(σ) = verdadero. Lo mismo, σ |≠ p abrevia S(p)(σ) = falso. P.ej., Si σ(x) = 1 y σ(y) = 2, entonces σ |= x < y

Una aserción (elemento sintáctico) representa un conjunto de estados (elemento semántico), el conjunto de todos los estados que satisfacen la aserción. 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 σ |= true y σ |≠ false)



Una especificación de un programa S es un par de aserciones (p, q) asociadas a la entrada y la salida de S, respectivamente. La aserción p es la precondición de S, denota al conjunto de estados iniciales de S. La aserción q es la postcondición de S, denota al conjunto de estados finales de S.

Por ejemplo, la especificación Φ = (x = X, x = 2X) es satisfecha por un programa que duplica su entrada x (un caso sería 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).

Métodos axiomáticos de verificación de programas

En los programas secuenciales se consideran básicamente dos propiedades, la correctitud parcial y la terminación, que en conjunto conforman la correctitud total.

- Un programa S es parcialmente correcto con respecto a una especificación (p, q) sii para todo estado σ:

  $$(\sigma \models p \land val(\pi(S, \sigma)) \neq \bot) \rightarrow val(\pi(S, \sigma)) \models q$$

  o sea, sii desde cualquier estado σ que satisface p, si S termina lo hace en un estado σ' que satisface q. La expresión |= {p} S {q} denota que S es parcialmente correcto con respecto a (p, q).

- Un programa S es totalmente correcto con respecto a una especificación (p, q) sii para todo estado σ:

  $$\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \bot \land val(\pi(S, \sigma)) \models q)$$

o sea, sii desde cualquier estado σ que satisface p, S termina en un estado σ' que satisface q.

La expresión |= ⟨p⟩ S ⟨q⟩ denota que S es totalmente correcto con respecto a (p, q). En particular, |= ⟨p⟩ S ⟨true⟩ denota que a partir de un estado que satisface p, S termina (en algún estado).

Las dos pruebas anterior mencionadas se basan en técnicas distintas.

La metodología de prueba de un programa S con respecto a una especificación (p, q) plantea:

Descomponer la prueba de correctitud total |= ⟨p⟩ S ⟨q⟩ en dos partes: (a) |= {p} S {q}, (b) |= ⟨p⟩ S ⟨true⟩.

a. Se usa el método de prueba de correctitud parcial H. Dicho método, compuesto por axiomas y reglas, permite probar sintácticamente la fórmula semántica |= {p} S {q}, lo que se expresa así: $|\!-_{H}$ {p} S {q}

b. Se usa el método de prueba de no divergencia H*. Es una extensión de H. Permite probar sintácticamente la fórmula semántica |= ⟨p⟩ S ⟨true⟩, lo que se expresa con: $|\!-_{H^*}$ ⟨p⟩ S ⟨true⟩.

- Se debe asegurar que los métodos de prueba sean sensatos:

  Si $|\!-_{H}$ {p} S {q}, entonces |= {p} S {q}

  Si $|\!-_{H^*}$ ⟨p⟩ S ⟨true⟩, entonces |= ⟨p⟩ S ⟨true⟩

- También que sean completos:

  Si |= {p} S {q}, entonces $|\!-_{H}$ {p} S {q}

  Si |= ⟨p⟩ S ⟨true⟩, entonces $|\!-_{H^*}$ ⟨p⟩ S ⟨true⟩

Lema de la forma de las computaciones

De la definición se infiere el determinismo de PLW: un programa tiene una sola computación, sólo una configuración sucede a otra. Además, se pueden desarrollar fácilmente las distintas formas que pueden tener las computaciones de los programas. Por ejemplo, en el caso de la secuencia, una computación π(S1 ; S2, σ) tiene tres formas posibles:

1. Una computación infinita $(S_1 ; S_2 , \sigma_0) \rightarrow (T_1 ; S_2 , \sigma_1) \rightarrow (T_2 ; S_2 , \sigma_2) \rightarrow \ldots$, cuando S_1 diverge a partir de σ_0 .

2. Otra computación infinita $(S_1 ; S_2 , \sigma_0) \rightarrow \ldots \rightarrow (S_2 , \sigma_1) \rightarrow (T_1 , \sigma_2) \rightarrow (T_2 , \sigma_3) \rightarrow \ldots$, cuando S_1 termina a partir de σ_0 y S_2 diverge a partir de σ_1 .

3. Una computación finita $(S_1 ; S_2 , \sigma_0) \rightarrow \ldots \rightarrow (S_2 , \sigma_1) \rightarrow \ldots \rightarrow (E, \sigma_2)$, cuando S_1 termina a partir de σ_0 y S_2 termina a partir de σ_1 .

De modo similar se pueden desarrollar las formas de las computaciones del resto de las instrucciones

Ejemplo 1. Computación de un programa.

- Sea el programa Sswap :: z := x ; x := y ; y := z, y el estado inicial σ0 , con σ0 (x) = 1 y σ0 (y) = 2.
- Utilizando la relación de transición "→" se prueba que Sswap intercambia los contenidos de las variables x e y:
  π(Sswap, σ0) = (z := x ; x := y ; y := z, σ0 [x|1][y|2]) →
  (x := y ; y := z, σ0 [x|1][y|2][z|1]) →
  (y := z, σ0 [y|2][z|1][x|2]) →
  (E, σ0 [z|1][x|2][y|1])
- Quedó: val(π(Sswap, σ0)) = σ1 , con σ1 (x) = 2 y σ1 (y) = 1.
- Generalizando, si σ0 (x) = X y σ0 (y) = Y, entonces val(π(Sswap, σ0)) = σ1 , con σ1 (x) = Y y σ1 (y) = X.

Ejemplo 3. Todo programa S cumple |= {true} S {true}. En palabras, a partir de cualquier estado, todo programa S, si termina, lo hace en algún estado.

La prueba es la siguiente. Sea un estado σ y un programa S. Debe cumplirse:

$(\sigma \models \text{true} \wedge \text{val}(\pi(S, \sigma)) \neq \perp) \rightarrow \text{val}(\pi(S, \sigma)) \models \text{true}$.

Se cumple $\sigma \models$ true. Si $\text{val}(\pi(S, \sigma)) = \perp$, entonces se cumple la implicación trivialmente.

(p = F, q = V, p → q = V)

Y si $\text{val}(\pi(S, \sigma)) \neq \perp$, entonces $\text{val}(\pi(S, \sigma)) = \sigma' \models$ true, por lo que también en este caso se cumple la implicación.

Ejemplo 4. ¿Todo programa S cumple |= <true> S <true>?

La respuesta es no, porque con que haya un estado inicial a partir del cual un determinado S no termina, no vale la fórmula.

Contraejemplo: un estado inicial con x = 0 y un programa que loopee a partir de x = 0.

Ejemplo 5. Si se cumple |= {true} S {false}, entonces significa que S no termina a partir de ningún estado. La prueba es la siguiente. Sea un estado σ y un programa S. Debe cumplirse:

(σ |= true ∧ val(π(S, σ)) ≠ ⊥) → val(π(S, σ)) |= false.

Se cumple σ |= true. Si val(π(S, σ)) ≠ ⊥, entonces val(π(S, σ)) = σ' |= false (absurdo). Por lo tanto val(π(S, σ)) = ⊥, es decir, el programa S no termina a partir de σ.

Ejemplo 6. Lema de Separación

Vale |= <p> S <q> ↔ (|= {p} S {q} ∧ |= <p> S <true>).

- Primero se probará |= <p> S <q> → (|= {p} S {q} ∧ |= <p> S <true>):

    Sea |= <p> S <q>. Entonces, dado σ, vale σ |= p → (val(π(S, σ)) ≠ ⊥ ∧ val(π(S, σ)) |= q). Por lo tanto:
    (a) (σ |= p ∧ val(π(S, σ)) ≠ ⊥) → val(π(S, σ)) |= q, es decir |= {p} S {q}.
    (b) σ |= p → (val(π(S, σ)) ≠ ⊥ ∧ val(π(S, σ)) |= true), es decir |= <p> S <true>.

    Así, por (a) y (b): |= {p} S {q} ∧ |= <p> S <true>.

- Ahora se probará (|= {p} S {q} ∧ |= <p> S <true>) → |= <p> S <q>:

    Sea |= {p} S {q} ∧ |= <p> S <true>. Entonces, dado σ, vale:
    (a) (σ |= p ∧ val(π(S, σ)) ≠ ⊥) → val(π(S, σ)) |= q.
    (b) σ |= p → (val(π(S, σ)) ≠ ⊥ ∧ val(π(S, σ)) |= true).
    (c) Por (b): σ |= p → val(π(S, σ)) ≠ ⊥.
    (d) Por (a) y (b): σ |= p → val(π(S, σ)) |= q.

    Así, por (c) y (d): σ |= p → (val(π(S, σ)) ≠ ⊥ ∧ val(π(S, σ)) |= q), es decir |= <p> S <q>.

Método axiomático de verificación de correctitud parcial H

Es para los programas con while.

H tiene axiomas y reglas que permiten probar sintácticamente la correctitud parcial de un programa S con respecto a una especificación (p, q). Se utiliza la expresión sintáctica |– H {p} S {q}.

Método H

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

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

p[x|e] expresa la sustitución en p de todas las ocurrencias libres de la variable x por la expresión 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)

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

6. Regla de consecuencia (CONS)

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

El axioma ASI establece que si se cumple p en términos de x después de la ejecución de una asignación x := e, significa que antes de la ejecución se cumple p en términos de e. Se lee "hacia atrás", de derecha a izquierda, lo que impone una forma de desarrollar las pruebas de H en el mismo sentido, de la postcondición a la precondición.

En la **regla de la secuencia (SEC),** el predicado (o aserción) r actúa como **nexo** y luego se descarta, no se propaga. Se permite usar la siguiente generalización:

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

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

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*.

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

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

- 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 manipular **todos los axiomas del dominio semántico** en las pruebas (en nuestro caso los axiomas de los números enteros, porque acotamos el estudio a dicho dominio).

- 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:

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

Recordar que antes probamos esto **semánticamente**, usando directamente la semántica operacional del lenguaje de programación. Ahora vamos a probarlo **sintácticamente**, mediante axiomas y reglas.

Por la forma del programa S_{swap}, recurrimos al axioma ASI tres veces, una por cada asignación, y al final completamos la prueba utilizando 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)

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

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

Debo ordenar la aplicación del axioma ASI de tal manera que me de pie para poder hacer SEC (generalmente sucede esto).

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

- Por la **sensatez** de H, que probaremos en otra clase, se cumple:

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

- Obviamente, **semánticamente** también se cumple la misma fórmula pero permutando los operandos de la precondición, es decir:

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

- Para probar esta última fórmula **sintácticamente** tenemos que recurrir a la regla CONS, agregando a la prueba anterior, que terminaba en:

  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)

La aserción del paso 5 es un axioma de los números enteros, por eso se justifica el paso con el indicador MAT (por matemáticas).

Si quisiéramos instanciarla, por ejemplo en: $\{y = 2 \wedge x = 1\}$ Sswap $\{y = 1 \wedge x = 2\}$ 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 $\{true\}\ S_{va}\ \{y \geq 0\}$

Considerando las dos asignaciones del programa, se propone que los primeros pasos de la prueba sean:

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

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

Para poder aplicar la regla COND, se necesitan dos fórmulas $p \wedge B$ y $p \wedge \neg B$, que en este caso tendrían la forma $p \wedge x > 0$ y $p \wedge \neg(x > 0)$.

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

Probamos con p = true, quedando entonces $true \wedge x > 0$, y $true \wedge \neg(x > 0)$, que de alguna manera deberemos relacionarlas con $x \geq 0$ y $-x \geq 0$ (continúa en el slide siguiente).

Entonces, partimos de:

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

hacemos:

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

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

y completamos la prueba de la siguiente manera:

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

- Respondiendo a la pregunta del slide anterior, efectivamente $(true, y \geq 0)$ **no** es una especificación correcta de un programa que calcula el valor absoluto. P.ej., el programa:

  S :: y := 1

  satisface $(true, y \geq 0)$ y no es el programa buscado. La variable *y* al final no tiene por qué tener el valor absoluto del contenido de la variable x al inicio. P.ej., se cumple: $\{x = 10\}$ y := 1 $\{y \neq |x|\}$). Una especificación correcta de un programa de valor absoluto sería:

  $(x = X, y = |X|)$

Axiomas y reglas adicionales

- Por la **completitud** de H (se prueba en otra clase), agregarle axiomas y reglas al método es **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 de H son:

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

  Cuando las variables 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,** con distintas precondiciones e iguales postcondiciones.

- Regla de la conjunción (AND) $$\frac{\{p_1\}\ S\ \{q_1\},\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

  Util para una **verificación por casos,** con distintas precondiciones y postcondiciones.

- 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**.

Proof Outlines

Se intercalan los pasos de la prueba entre las instrucciones del programa. Se obtiene una prueba más estructurada, que documenta adecuadamente el programa.

En los programas concurrentes son imprescindibles.

La siguiente es una proof outline de correctitud parcial de un programa que calcula el factorial:

```
{x > 0}
S_fac :: 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!}
```

Toda aserción usada en una prueba es en realidad un **invariante**. Volviendo a la proof outline anterior: cualquiera sea el estado inicial, toda aserción 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 invariante se cumple a lo largo de toda la computación del while.

La correctitud parcial pertenece a la familia de las propiedades safety. Son propiedades que se prueban por inducción.

Ejemplo 1. Prueba de correctitud parcial de un programa que efectúa la división entera entre dos números enteros

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

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

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

Recordatorio de la regla REP

La dificultad de la verificación de un programa radica fundamentalmente en encontrar aserciones en distintas locaciones, en particular los **invariantes** de los *while*, para asociarlas con las premisas de alguna regla y obtener la conclusión deseada (de todos modos recordar que estudiamos pruebas a posteriori para simplificar la exposición, siendo en cambio la buena práctica **construir un programa en simultáneo con su verificación**).

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:

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

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)

La completitud se asocia a la problemática de la expresividad de las especificaciones que depende del lenguaje de programación y de la interpretación semántica de las variables. En nuestro caso, que trabajamos con el lenguaje de la lógica de predicados, los programas con while y los números enteros, se cumple la expresividad.

Otra causal de incompletitud es la interpretación semántica de las variables. Trabajando con variables enteras se sabe que hay enunciados de los enteros que no se pueden probar con ninguna axiomática. El método H debe entenderse como que incluye al conjunto de todos los axiomas de los números enteros. Su completitud no es absoluta, es relativa