Clase teórica 12

Sensatez y completitud de la Lógica de Hoare

Introducción

- Vamos a demostrar la **sensatez** y la **completitud** de los métodos de prueba **H y H*.**

- Formalmente, si:

  $\vdash_H \{p\}\ S\ \{q\}$ expresa que H permite probar **sintácticamente** {p} S {q}

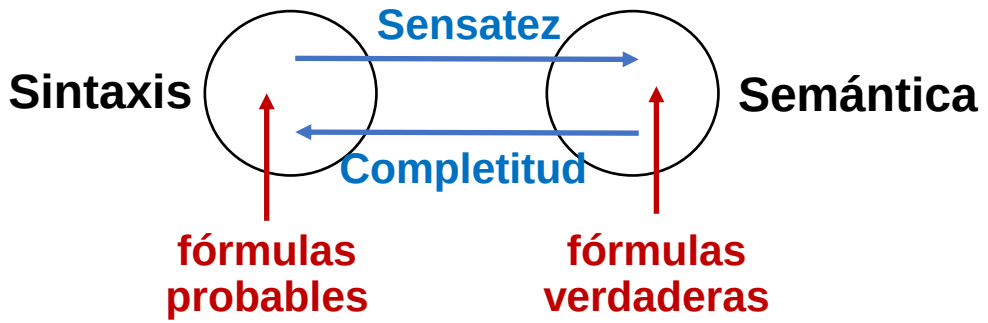
  $\vdash_{H^*} \langle p\rangle\ S\ \langle q\rangle$ expresa que H* permite probar **sintácticamente** ⟨p⟩ S ⟨q⟩

  $\models \{p\}\ S\ \{q\}$ expresa que se cumple **semánticamente** {p} S {q} (correctitud parcial de S con respecto a (p, q))

  $\models \langle p\rangle\ S\ \langle q\rangle$ expresa que se cumple **semánticamente** ⟨p⟩ S ⟨q⟩ (correctitud total de S con respecto a (p, q))

  vamos a demostrar que para todo programa S y para toda especificación (p, q) se cumplen las implicaciones:

  1. $\mathbf{\vdash_H \{p\}\ S\ \{q\} \rightarrow \models \{p\}\ S\ \{q\}}$ (sensatez de H)
  2. $\mathbf{\models \{p\}\ S\ \{q\} \rightarrow \vdash_H \{p\}\ S\ \{q\}}$ (completitud de H)
  3. $\mathbf{\vdash_{H^*} \langle p\rangle\ S\ \langle q\rangle \rightarrow \models \langle p\rangle\ S\ \langle q\rangle}$ (sensatez de H*)
  4. $\mathbf{\models \langle p\rangle\ S\ \langle q\rangle \rightarrow \vdash_{H^*} \langle p\rangle\ S\ \langle q\rangle}$ (completitud de H)



- En palabras:
  - Las fórmulas que prueban H y H* son **verdaderas** (sensatez).
  - Todas las fórmulas verdaderas **se pueden proba**r en H y H* (completitud).

- Antes repasaremos la técnica de **inducción**, y definiremos formalmente la **semántica de los lenguajes**.

Inducción matemática

- Sea P una propiedad a probar en el dominio $\mathcal{N}$ de los números naturales.

- Si

  1) **P(0)**

  2) **Para todo k de $\mathcal{N}$: P(k) ⟶ P(k + 1)**

  entonces

  3) **Para todo n de $\mathcal{N}$: P(n)**

- (1) es la **base inductiva**

  (2) es el **paso inductivo**

  P(k) es la **hipótesis inductiva**

- Variante fuerte de la inducción matemática

  Paso inductivo: $(P(i) \wedge P(i + 1) \wedge \ldots \wedge P(k - 1) \wedge P(k)) \longrightarrow P(k + 1)$, para algún i

Ejercicio: probar que para todo n de $\mathcal{N}$ se cumple: $0 + 1 + 2 + 3 + \ldots + n = (n \cdot n + 1) / 2$

Inducción estructural

- Generalización de la inducción matemática.
- Util también para definir conjuntos.
- Se aplica a cualquier dominio, no sólo $\mathcal{N}$, considerando una determinada relación menor <.
- Puede haber varios minimales, varias bases inductivas y varios pasos inductivos.

- **Ejemplo**

  Definición por inducción estructural de las expresiones aritméticas con símbolos de {0 , 1 , x , + , . , (,)}:

  (a) bases inductivas: 0, 1, x, son expresiones aritméticas.

  (b) pasos inductivos: si e_1 y e_2 son expresiones aritméticas, también lo son $(e_1 + e_2)$ y $(e_1 . e_2)$.

Ejercicio: probar C(e) = 1 + O(e), con:

C(e) = constantes y variables de e
O(e) = operadores de e

Ejemplo: e = (1 + (x . 0))

C(e) = 3
O(e) = 2

Semántica operacional del lenguaje de programación

- **Idea general.** Dados un programa S y un estado inicial σ, a dicha **configuración inicial** (S, σ) se le asocia una **computación** π(S, σ), que es la secuencia de configuraciones de la ejecución de S a partir de σ.
  P.ej.: (x := 0 ; y := 1 ; z := 2, σ) → (y := 1 ; z := 2, σ[x|0]) → (z := 2, σ[x|0][y|1]) → (E, σ[x|0][y|1][z|2]).

- **Notación.** val(π(S, σ)) es el estado final de π(S, σ). Si π(S, σ) es infinita, val(π(S, σ)) = ⊥.
  P.ej.: val(π(while true do skip od, σ)) = ⊥, y val(π(x := 10 ; while x > 0 do x := x – 1 od, σ)) = σ[x|0].

- **Semántica de las instrucciones.** Se define por inducción estructural, mediante una relación → de **transición de configuraciones** (pares (S, σ) con **continuación sintáctica** S y estado σ):

  1. (skip, σ) → (E, σ)
     El skip se consume en un paso (es atómico) y no modifica el estado inicial. E es la continuación sintáctica vacía.
  2. (x := e, σ) → (E, σ[x|σ(e)])
     La asignación también es atómica y el estado final es como el inicial salvo que el valor de la variable x es el de la expresión e según σ.
  3. Para toda instrucción T, si (S, σ) → (S', σ'), entonces (S ; T, σ) → (S' ; T, σ')
     La secuencia S ; T se ejecuta de izquierda a derecha. Una vez consumido S, si no diverge, se ejecuta T. Se define E ; S = S ; E = S.
  4. (if B then S_1 else S_2 fi, σ) → (S_1, σ), si se cumple σ(B) = V.
     (if B then S_1 else S_2 fi, σ) → (S_2, σ), si se cumple σ(B) = F.
     La evaluación de la expresión B es atómica y no modifica el estado inicial. Su evaluación determina si se ejecuta S_1 o S_2.
  5. (while B do S od, σ) → (S ; while B do S od, σ), si se cumple σ(B) = V.
     (while B do S od, σ) → (E, σ), si se cumple σ(B) = F.
     La evaluación de B es atómica y no modifica el estado inicial. Su evaluación determina si se ejecuta S o se termina la repetición.

- **Nota**

  Las expresiones σ(e) y σ(B) son *abusos de notación* para simplificar,
  porque un estado σ se define sobre variables, no sobre expresiones.
  En el Anexo se define su semántica.

- **Definición por extensión de la semántica de las instrucciones**

  De la definición por **comprensión** de las instrucciones se deriva su definición por **extensión**.

  Por ejemplo, en el caso de la **secuencia**, una computación $\pi(S_1 ; S_2, \sigma)$ 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. Una 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 obtener las formas de las computaciones de las otras instrucciones.

Ejercicio. Obtener las formas de las computaciones de las otras instrucciones.

Sensatez del método H

- Se cumple, para todo programa S y toda especificación (p, q), que: **$\vdash_H$ {p} S {q} $\rightarrow$ |= {p} S {q}.**

- Lo probaremos por inducción matemática fuerte, considerando la **longitud de la prueba** (1 o más pasos):
  (a) Base inductiva: los **axiomas** son verdaderos (pruebas de tamaño 1).
  (b) Paso inductivo: las **reglas** son sensatas, preservan la verdad de las premisas (pruebas de tamaño ≥ 2).

- Ejemplo de (a) - el resto queda como ejercicio -. Prueba de que el axioma **SKIP** es verdadero.

  - Dado |– {p} skip {p}, hay que probar **|= {p} skip {p}.**
  - Por la semántica del lenguaje: (skip, σ) $\rightarrow$ (E, σ).
  - Sea **σ |= p.** Luego del skip se cumple **σ |= p.**

- Ejemplo de (b) - el resto queda como ejercicio -. Prueba de que la regla **SEC** es sensata.

  - Dado |– {p} S_1 ; S_2 {q}, hay que probar **|= {p} S_1 ; S_2 {q}.**
  - |– {p} S_1 ; S_2 {q} se obtiene de **|– {p} S_1 {r} y |– {r} S_2 {q}** (pruebas más cortas que |– {p} S_1 ; S_2 {q}).
  - Entonces, por hipótesis inductiva: **|= {p} S_1 {r}** y **|= {r} S_2 {q}.** Veamos que se cumple **|= {p} S_1 ; S_2 {q}:**
  - Sea **σ_1 |= p,** y asumamos que S_1 ; S_2 termina desde σ_1 (si no temina, la terna se cumple trivialmente).
  - Por la semántica del lenguaje y la hipótesis inductiva: (S_1 ; S_2, σ_1) $\rightarrow$ … (S_2, σ_2) $\rightarrow$ … (E, σ_3), tal que se cumple σ_2 |= r y **σ_3 |= q,** que es lo que queríamos demostrar.

Completitud del método H

- Se cumple, para todo programa S y toda especificación (p, q), que **$\models \{p\}\ S\ \{q\} \rightarrow \vdash_H \{p\}\ S\ \{q\}$.**

- Lo probaremos por inducción estructural, considerando las **5 formas que pueden tener los programas.**
  (a) Base inductiva: considera los **programas atómicos:** skip y x := e.
  (b) Paso inductivo: considera los **programas compuestos:** secuencia, if then else y while.

- Ejemplo de (a). Prueba de que: **$\models \{p\}$ skip $\{q\} \rightarrow \vdash \{p\}$ skip $\{q\}$.**

  - Se tiene **$\models \{p\}$ skip $\{q\}$.**
  - Por la semántica del skip vale $\models \{p\}$ skip $\{p\}$, por lo que debe cumplirse **$p \rightarrow q$**.
  - De esta manera se logra: **$\vdash \{p\}$ skip $\{q\}$:**

    1. $\{p\}$ skip $\{p\}$ (SKIP)
    2. $p \rightarrow q$ (MAT)
    3. $\{p\}$ skip $\{q\}$ (CONS, 1, 2)

- Ejemplo de (b). Prueba de que: **$\models \{p\}$ if B then S_1 else S_2 fi $\{q\} \rightarrow \vdash \{p\}$ if B then S_1 else S_2 fi $\{q\}$.**

  - Se tiene **$\models \{p\}$ if B then S_1 else S_2 fi $\{q\}$.**
  - Por la semántica del if vale $\models \{p \wedge B\}\ S_1\ \{q\}$ y $\models \{p \wedge \neg B\}\ S_2\ \{q\}$ (S_1 y S_2 son más simples que todo el if).
  - Entonces, por hipótesis inductiva: **$\vdash \{p \wedge B\}\ S_1\ \{q\}$** y **$\vdash \{p \wedge \neg B\}\ S_2\ \{q\}$.**
  - De esta manera, aplicando la regla COND, se logra: **$\vdash \{p\}$ if B then S_1 else S_2 fi $\{q\}$.**

- Otro ejemplo de (b). Prueba de que: **$\models \{p\}\ S_1 ; S_2\ \{q\} \rightarrow \vdash \{p\}\ S_1 ; S_2\ \{q\}$:**
  - Se tiene **$\models \{p\}\ S_1 ; S_2$ fi $\{q\}$.**
  - Por la semántica de la secuencia vale $\models \{p\}\ S_1\ \{r\}$ y $\models \{r\}\ S_2\ \{q\}$ (S_1 y S_2 son más simples que $S_1 ; S_2$).
  - Entonces, por hipótesis inductiva: **$\vdash \{p\}\ S_1\ \{r\}$** y **$\vdash \{r\}\ S_2\ \{q\}$.**
  - De esta manera, aplicando la regla SEC, se logra: **$\vdash \{p\}\ S_1 ; S_2\ \{q\}$**.

¿Pero se puede asegurar que existe un predicado r expresable en la lógica de predicados?

- En este caso sí, porque la lógica de predicados es **expresable** con respecto al lenguaje de programación y el dominio de los números enteros (no tiene por qué cumplirse en otros casos):

Para todo predicado p y todo programa S, siempre se puede expresar la postcondición q de {p} S {q}

- Lo mismo ocurre en la prueba de $\models \{p\}$ while B do S od $\{q\} \rightarrow \vdash \{p\}$ while B do S od $\{q\}$:

Siempre se pueden expresar los invariantes.

- Pero H no es completo en términos **absolutos**:
  - La axiomática de los números enteros es incompleta (Teorema de Gödel).
  - Por lo tanto, hay que asumir que H tiene todos los axiomas de los enteros.
  - P.ej., si vale $\models \{true\}$ skip $\{p\}$, para probar $\vdash \{true\}$ skip $\{p\}$ hay que asumir que p es un axioma de H.
  - De todos modos, de lo que se trata es de probar programas, no enunciados de los números enteros.

¿Impacta en la automatización que la completitud de H no sea absoluta?

Sensatez del método H*

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

regla REP*

- Sólo falta probar que la regla REP* es sensata:

  $\vdash_{H^*} \langle p\rangle$ while B do S od $\langle p \wedge \neg B\rangle \rightarrow \models \langle p\rangle$ while B do S od $\langle p \wedge \neg B\rangle$

Inducción matemática fuerte

Sea: **$\vdash \langle p\rangle$ while B do S od $\langle p \wedge \neg B\rangle$**

Entonces: $\vdash \langle p \wedge B\rangle\ S\ \langle p\rangle$, $\vdash \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle$, $p \rightarrow t \geq 0$

Por hip. ind.: **$\models \langle p \wedge B\rangle\ S\ \langle p\rangle$, $\models \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle$, $p \rightarrow t \geq 0$** (S es más simple que while B do S od)

Veamos que: **$\models \langle p\rangle$ while B do S od $\langle p \wedge \neg B\rangle$**

Sea: **$\sigma_0 \models p$**

Hay que probar que el while termina desde σ_0 en un estado **$\sigma_k \models p \wedge \neg B$**

Supondremos que no, y llegaremos a una contradicción:

Como: **(while B do S od, σ_0) $\rightarrow$.. (while B do S od, σ_1) $\rightarrow$.. (while B do S od, σ_2) $\rightarrow$.., con $\sigma_i \models ((p \wedge B) \wedge t \geq 0)$**

Entonces: **$\sigma_0(t)$ > $\sigma_1(t)$ > $\sigma_2(t)$ > …**

lo que es absurdo porque esta cadena es una cadena descendente infinita de números naturales.

Por lo tanto, el while termina y lo hace en un estado $\sigma_k \models p \wedge \neg B$

Completitud del método H*

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

regla REP*

- Sólo falta probar que la regla REP* es completa:

  $\models \langle p\rangle$ while B do S odS $\langle q\rangle \rightarrow \vdash_{H^*} \langle p\rangle$ while B do S od $\langle q\rangle$

<u>Inducción estructural</u>

La idea central es probar la **expresividad** del **invariante** y el **variante**.

La expresividad del invariante ya la comentamos en la prueba sobre H (ver más detalle en el Anexo).

Veamos el caso del **variante**:

Hay que definir una **función entera t** que exprese **la cantidad de iteraciones del while:**

Sea **S :: while B do T od** y un estado inicial **σ.**

Sea **iter(S, σ)** una función parcial que especifica la cantidad de iteraciones de S a partir de σ.

La función iter(S, σ) es **computable**:

S_x :: x := 0 ; while B do x := x + 1 ; T od

Si S termina en un estado σ´, **iter(S, σ) = σ´(x)** contiene la cantidad de iteraciones del while.

En definitiva, el lenguaje de especificación debe poder expresar **todas las funciones computables.**

Anexo

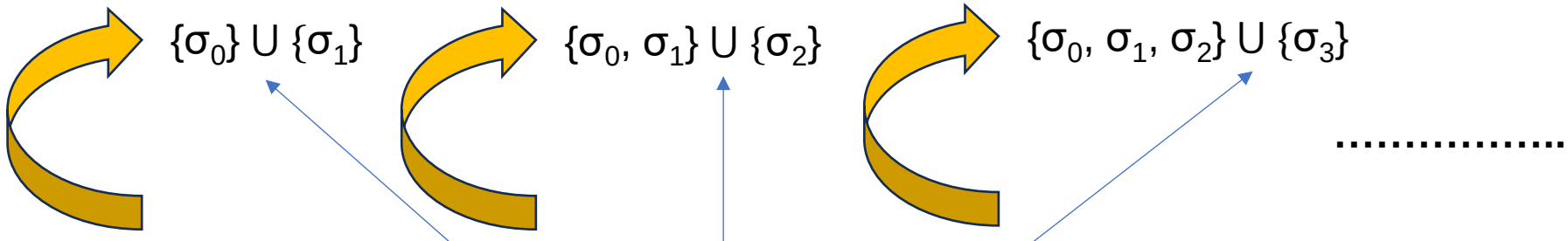
Expresión del invariante de un while

- Dado: **{r} while B do S od {q}**

  hay que expresar con un invariante p un conjunto C con todos los estados alcanzables por el *while* desde $\sigma_0 \models r$

$$C = \{\sigma \mid \exists k, \sigma_0, \ldots, \sigma_k : k \in \mathcal{N} \wedge \sigma_0 \models r \wedge \sigma = \sigma_k \wedge (\forall i < k: M(S)(\sigma_i) = \sigma_{i+1} \wedge \sigma_i(B) = V)\}$$

$\sigma_0 \models r$
while B
do
S
od



- Sintácticamente, el invariante p se podría plantear como la disyunción, posiblemente infinita:

$$r \vee p_1 \vee p_2 \vee p_3 \vee \ldots$$

donde p_{i+1} expresa la postcondición de S a partir de $p_i \wedge B$ (con $r = p_0$)

pero dicha disyunción no es un predicado válido.

Se prueba que existe una expresión finita para el invariante p.

Semántica funcional (o denotacional) de las expresiones y predicados

- **Expresiones enteras**

$$\mathbf{e :: n \mid x \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 . e_2) \mid \ldots}$$

n es una constante entera. x es una variable entera. P.ej: **e = (5 + (x – 1))**

- Se utiliza una **función semántica V**, que asigna a una **expresión entera e** y un **estado σ** un **número entero** n

$$\mathbf{V : Iexp \longrightarrow (\Sigma \longrightarrow Z)}$$

Iexp es el conjunto de las expresiones enteras

- Definición inductiva

1. V(n)(σ) = n, tal que n es la representación del número entero n (p.ej., el símbolo 1 representa el número 1)
2. V(x)(σ) = σ(x)
3. $V((e_1 + e_2))(\sigma) = V(e_1)(\sigma) + V(e_2)(\sigma)$
4. $V((e_1 - e_2))(\sigma) = V(e_1)(\sigma) - V(e_2)(\sigma)$

Etc.

- **Expresiones booleanas**

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

true y false son las constantes booleanas. Las e_i son expresiones enteras. P.ej: **$B = ((x = 1) \vee (x = 2))$**

- Se utiliza una **función semántica W**, que asigna a una **expresión booleana B** y un **estado σ** un **valor V o F**

$$\mathbf{W : Bexp \longrightarrow (\Sigma \longrightarrow \{V, F\})}$$

Bexp es el conjunto de las expresiones booleanas

- Definición inductiva

1. $W(true)(\sigma) = V$ y $W(false)(\sigma) = F$ (p.ej., la constante true representa el valor de verdad V)
2. $W((e_1 = e_2))(\sigma) = (V(e_1)(\sigma) = V(e_2)(\sigma))$
3. $W((e_1 < e_2))(\sigma) = (V(e_1)(\sigma) < V(e_2)(\sigma))$
4. $W(\neg B_1)(\sigma) = \neg W(B_1)(\sigma)$
5. $W((B_1 \vee B_2))(\sigma) = W(B_1)(\sigma) \vee W(B_2)(\sigma)$

Etc.

- **Predicados**

$$\mathbf{p :: true \mid false \mid (e_1 = e_2) \mid (e_1 < e_2) \mid \ldots \mid \neg p \mid (p_1 \vee p_2) \mid \ldots \mid \exists x{:}\ p \mid \forall x{:}\ p}$$

true y false son las constantes. x es una variable entera. Las e_i son expresiones enteras. P.ej: $\exists x{:}\ \neg(x = 1)$

- Se utiliza una **función semántica T**, que asigna a un **predicado p** y un **estado σ** un **valor V o F**

$$\mathbf{T : Pred \longrightarrow (\Sigma \longrightarrow \{V, F\})}$$

Pred es el conjunto de los predicados

- Definición inductiva

1. $T(true)(\sigma) = V$ y $T(false)(\sigma) = F$ (p.ej., la constante true representa el valor de verdad V)
2. $T((e_1 = e_2))(\sigma) = (T(e_1)(\sigma) = T(e_2)(\sigma))$. Y lo mismo con los otros operadores (<, >, etc.)
3. $T(\neg p_1)(\sigma) = \neg T(p_1)(\sigma)$. Y lo mismo con los otros operadores ($\wedge$, $\vee$, etc.)
4. $T(\exists x{:}\ p_1)(\sigma) = V$ sii existe algún número entero n tal que $T(p_1)(\sigma[x|n]) = V$
5. $T(\forall x{:}\ p_1)(\sigma) = V$ sii para todo número entero n se cumple $T(p_1)(\sigma[x|n]) = V$

Semántica funcional (o denotacional) de las instrucciones

- Se utiliza una función semántica M, que asigna a un programa S y un estado σ otro estado σ.

$$\mathbf{M : L_w \longrightarrow (\Sigma \longrightarrow \Sigma)}$$

L_w es el lenguaje de programación

- Definición inductiva

1. $M(skip)(\sigma) = \sigma$
2. $M(x := e)(\sigma) = \sigma[x|V(e)(\sigma)]$
3. $M(S_1 ; S_2)(\sigma) = M(S_2)(M(S_1)(\sigma))$
4. $M(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi})(\sigma) = M(S_1)(\sigma)$, si $W(B)(\sigma) = V$
   $= M(S_2)(\sigma)$, si $W(B)(\sigma) = F$
5. $M(\text{while } B \text{ do } S \text{ od})(\sigma) = M(\text{while } B \text{ do } S \text{ od})(M(S)(\sigma))$, si $W(B)(\sigma) = V$
   $= \sigma$, si $W(B)(\sigma) = F$

con $M(S)(\sigma) = \perp$ si S diverge, y $M(S)(\perp) = \perp$ para todo S

- **La especificación así planteada está incompleta**, falta encontrar la solución de **M(while B do S od)(σ)**

  - Una manera es mediante el *límite* de una secuencia de funciones conocidas como *aproximaciones*, que representan las sucesivas iteraciones del *while*.
  - Para asegurar su *existencia* y *unicidad* se recurre a la **teoría de los *órdenes parciales completos***.

Semántica funcional (o denotacional) del while

- Resolución de la ecuación **M(while B do S od)(σ) = M(while B do S od)(M(S)(σ)), si W(B)(σ) = V**
  = σ, si W(B)(σ) = F

- Se plantea la siguiente secuencia de funciones φ_i:

  1. $\varphi_0(\sigma) = \perp$
  2. $\varphi_{i+1}(\sigma) = \varphi_i(M(S)(\sigma))$, si $W(B)(\sigma) = V$

     σ, si $W(B)(\sigma) = F$

  y se calcula el **límite φ de la cadena $\{\varphi_i\}_{i \geq 0}$**

- Por ejemplo, dado el while: **S :: while x > 0 do x := x – 1 od** y el estado inicial **σ[x|2]**:

  $\varphi_0(\sigma[x|2]) = \perp$

  $\varphi_1(\sigma[x|2]) = \varphi_0(\sigma[x|1]) = \perp$

  $\varphi_2(\sigma[x|2]) = \varphi_1(\sigma[x|1]) = \varphi_0(\sigma[x|0]) = \perp$

  $\varphi_3(\sigma[x|2]) = \varphi_2(\sigma[x|1]) = \varphi_1(\sigma[x|0]) = \sigma[x|0]$

  $\varphi_i(\sigma[x|2]) = \sigma[x|0]$, con i = 4, 5, 6, …, es decir que el límite (estado final) es **σ[x|0]**.

Semántica operacional vs semántica funcional (o denotacional)

Semántica operacional

- Definida con computaciones, configuraciones, estados, etc., de una máquina.
- Muy intuitiva.
- Sirve como guía de implementación (el nivel de abstracción elegido es muy importante).

Semántica funcional (o denotacional)

- Definida con conjuntos, funciones, funcionales, etc. Se relacionan estructuras sintácticas con objetos matemáticos. Toda estructura **denota** un valor.
- No tiene el riesgo del nivel de abstracción.
- Su uso se dificulta cuando el lenguaje es muy complejo, sobre todo concurrente.

En definitiva, las dos semánticas son complementarias más que alternativas.

Tipos de sensatez

- La sensatez de H se cumple **independientemente de la interpretación semántica considerada**. Notar que en ningún momento de las pruebas recurrimos al dominio de las variables de programa (en nuestro caso los números enteros), salvo cuando utilizamos sus axiomas (que podrían ser de otro dominio semántico). Por ejemplo, en el programa de división presentado antes, **la prueba también aplica si las variables son de tipo real.**

- En efecto, H tiene **sensatez total**, lo que se formula de la siguiente manera:

$$\mathbf{Tr_I \mid\!-_H \{p\}\ S\ \{q\} \rightarrow \mid\!=_I \{p\}\ S\ \{q\},\ para\ toda\ interpretación\ I}$$

Tr_I tiene todos los axiomas correspondientes a la interpretación I (números enteros, números reales, etc).

Que valga $\mid\!=_I \{p\}\ S\ \{q\}$ significa que $\{p\}\ S\ \{q\}$ es verdadera considerando I.

- En cambio, la sensatez de H* se conoce como **aritmética**. Tal como está planteada la regla REP*:
  - El dominio semántico en el que se interpretan los programas **debe extenderse con el de los números naturales**, en el que está definido el variante t (en la hoja siguiente mostramos un ejemplo con números reales).
  - Correspondientemente, el lenguaje de especificación **debe extenderse con el de la aritmética**.

Ejemplo de prueba de terminación en el dominio de los números reales

- Apartándonos por un momento del dominio semántico de los números enteros, sea el siguiente programa S en el que las variables x y Ɛ son de tipo **real,** inicialmente mayores que 0:

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

- Claramente se cumple: **⟨x = X ∧ X > 0 ∧ Ɛ > 0⟩ S ⟨true⟩.**

- Los distintos valores positivos de x constituyen iteración tras iteración una secuencia decreciente estricta.

- También en este caso se puede usar la regla REP*, definiendo un adecuado variante t en el dominio de los números naturales.

- Una posible función t podría ser la siguiente:

$$t = \text{if } x > Ɛ \text{ then } [\log_2 X/Ɛ] - \log_2 X/x \text{ else } 0 \text{ fi}$$

Ejercicio: calcular los distintos valores de t a partir de X = 10 y Ɛ = 0,1.

Verificación de programas con arreglos y punteros

- El axioma de asignación (ASI) presentado antes no es verdadero cuando se agregan **arreglos** al lenguaje:

  1) {true} a[i] := 1 {a[i] = 1}

     Si **i = "a[2]", a = [2, 2, …],** luego de a[i] := 1 vale a[i] = 2.

     Motivo: en la asignación, a[i] se refiere a a[2], pero en la postcondición, a[i] se refiere a a[1].

  2) {0 + 1 = a[y]} a[x] := 0 {a[x] + 1 = a[y]}

     Si **a = [1, …], x = 1, y = 1,** luego de a[x] := 0 vale a[x] = 0, a[y] = 0.

     Motivo: a[x] y a[y] denotan un mismo elemento, son *alias*.

  Remediación: evitar variables con índices de la forma a[i], y alias, pero esto es muy restrictivo. Otra posibilidad es modificar el mecanismo de sustitución sintáctica, que es la base de la semántica de :=.

- Otro tipo de dato problemático en el marco de la axiomática definida es el **puntero**. En este caso un enfoque muy difundido es la **lógica de separación**.

Aún con variables enteras simples deben tomarse recaudos. La aritmética de las computadoras no es la de la matemática. P.ej., en caso de *overflow*, ¿se cancela el programa?, ¿se devuelve el máximo entero?, ¿se usa aritmética modular? **La axiomática debe contemplar la implementación adoptada**.

Clase práctica 12

Ejemplo 1. Computación de un programa.

- Sea el programa S_{swap} :: z := x ; x := y ; y := z, y el estado inicial σ_0, con $\sigma_0(x) = 1$ y $\sigma_0(y) = 2$.

- Utilizando la relación de transición $\rightarrow$ se prueba que S_{swap} intercambia los contenidos de las variables x e y:

$$\pi(S_{swap}, \sigma_0) = (z := x ; x := y ; y := z, \sigma_0[x|1][y|2]) \rightarrow$$
$$(x := y ; y := z, \sigma_0[x|1][y|2][z|1]) \rightarrow$$
$$(y := z, \sigma_0[y|2][z|1][x|2]) \rightarrow$$
$$(E, \sigma_0[z|1][x|2][y|1])$$

- Quedó: $val(\pi(S_{swap}, \sigma_0)) = \sigma_1$, con $\sigma_1(x) = 2$ y $\sigma_1(y) = 1$.

- Generalizando, si $\sigma_0(x) = X$ y $\sigma_0(y) = Y$, entonces $val(\pi(S_{swap}, \sigma_0)) = \sigma_1$, con $\sigma_1(x) = Y$ y $\sigma_1(y) = X$.

En programas simples como éste, la **verificación semántica** resulta factible, pero en programas complejos se torna **prohibitiva**.

Ejemplo 2. 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:

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

Se cumple σ |= true. Si val(π(S, σ)) = ⊥, entonces se cumple la implicación trivialmente.
Y si val(π(S, σ)) ≠ ⊥, entonces val(π(S, σ)) = σ' |= true, por lo que también en este caso se cumple la implicación.

Ejemplo 3. ¿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 4. 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:

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

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 5. Sensatez de la Regla de la Disyunción (OR)

La regla OR establece, dadas aserciones p, q, r, y un programa S:

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

Probaremos la sensatez de la regla OR por inducción sobre la longitud de la prueba. Hay que probar:

$\vdash \{p \vee r\}\ S\ \{q\} \rightarrow \models \{p \vee r\}\ S\ \{q\}$, habiendo aplicado OR

- $\vdash \{p \vee r\}\ S\ \{q\}$ proviene de $\vdash \{p\}\ S\ \{q\}$ y $\vdash \{r\}\ S\ \{q\}$, aplicando OR.
- Hipótesis inductiva: (1) $\models \{p\}\ S\ \{q\}$, (2) $\models \{r\}\ S\ \{q\}$.
- Veamos que se cumple **$\models \{p \vee r\}\ S\ \{q\}$**:
- Sea **$\sigma \models p \vee r$**, y asumamos que S termina desde σ en un estado σ´.
- Hay dos posibilidades, lo que completa la prueba:

  Si $\sigma \models p$, entonces por (1) vale **$\sigma' \models q$.**

  Si $\sigma \models r$, entonces por (2) vale **$\sigma' \models q$.**

Ejemplo 6. No sensatez de la siguiente regla UNTIL

$$\frac{\{p \wedge \neg B\}\ S\ \{p\}}{\{p\}\ \text{repeat } S \text{ until } B\ \{p \wedge B\}}$$

La semántica de la instrucción repeat es:

repeat S until B = S ; while ¬ B do S od

Veamos que la regla **no es sensata**, mediante un contraejemplo.

Supongamos **|= {p ∧ ¬ B} S {p}** (hipótesis inductiva).

A partir de ella, **encontraremos un caso en que no valga la fórmula |= {p} repeat S until B {p ∧ B}.**

La idea es tener en cuenta que la premisa de la regla asegura que S preserva p **sólo a partir de ¬B**. Efectivamente, podría suceder que **a partir de p ∧ B, S no preserve p**.

Veamos el desarrollo de la prueba en el slide siguiente:

Sean:

p = (x = 0)

B = (y = 0)

S :: if ¬(y = 0) then skip else x := 1 fi (notar que S no preserva p cuando vale B)

Veamos que se cumple |= {p ∧ ¬ B} S {p} pero que no se cumple |= {p} repeat S until B {p ∧ B}:

1. Se cumple |= {p ∧ ¬ B} S {p}:

|= {x = 0 ∧ ¬(y = 0)} if ¬(y = 0) then skip else x := 1 fi {x = 0}

$$\frac{\{p \wedge \neg B\}\ S\ \{p\}}{\{p\}\ \text{repeat } S \text{ until } B\ \{p \wedge B\}}\ a$$

2. No se cumple |= {p} repeat S until B {p ∧ B}:

Es decir, no se cumple |= {x = 0} repeat if ¬(y = 0) then skip else x := 1 fi until y = 0 {x = 0 ∧ y = 0}:

Si el estado inicial σ cumple: |= (x = 0 ∧ y = 0), entonces el repeat se ejecuta sólo una vez, y el estado final σ' cumple: σ' |= (x = 1 ∧ y = 0).