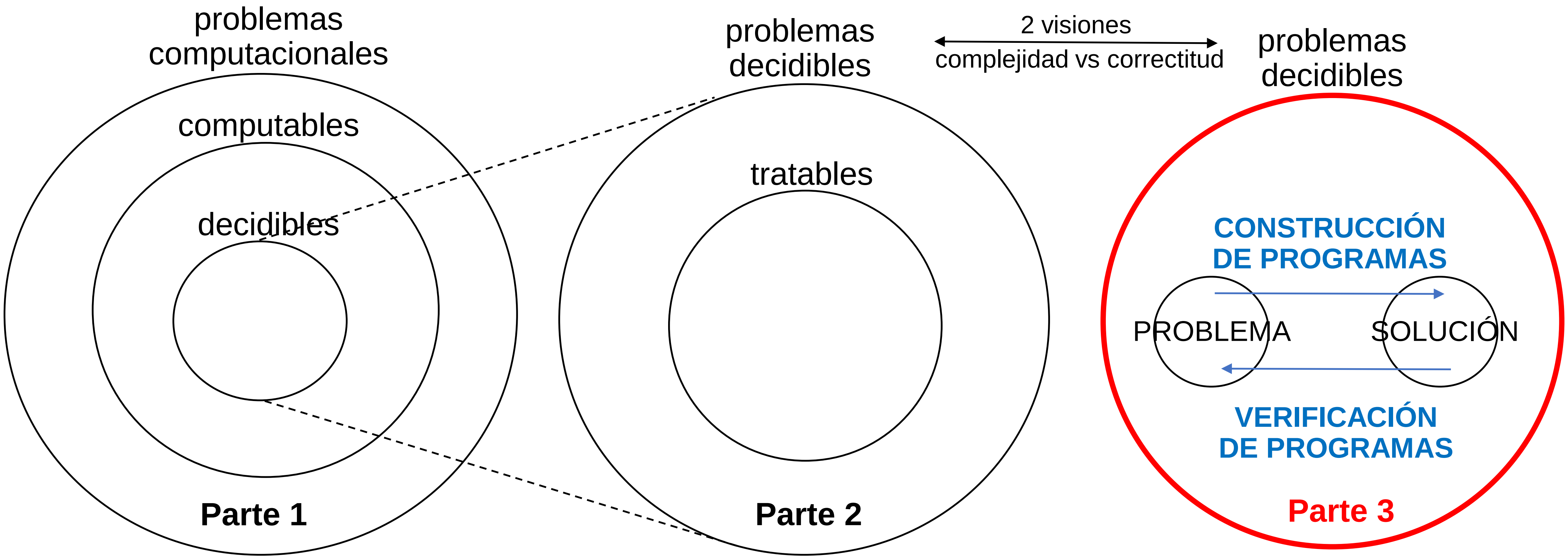
Clase teórica 10

Introducción a la Verificación de Programas

Introducción



Computabilidad
(años 1930)
Máquinas de Turing

Complejidad Computacional
(años 1960)
Máquinas de Turing

Correctitud de Programas
(años 1970)
Programas Secuenciales

Bibliografía

Básica

- Teoría de la Computación y Verificación de Programas. Rosenfeld & Irazábal. McGraw Hill y EDULP. 2010. Libro físico (en Biblioteca).

- Computabilidad, Complejidad Computacional y Verificación de Programas. Rosenfeld & Irazábal. EDULP. 2013. http://sedici.unlp.edu.ar/handle/10915/27887.

- Lógica para Informática. Pons, Rosenfeld & Smith. EDULP. 2017. http://sedici.unlp.edu.ar/handle/10915/61426.

- Verificación de Programas. Programas Secuenciales y Concurrentes. EDULP. 2024. Preliminar (en IDEAS).

Complementaria mínima (en Biblioteca)

- Program Verification. Nissim Francez. Addison-Wesley. 1992.

- Verification of Sequential and Concurrent Programs. Apt y Olderog. Springer. 1997.

Primeros conceptos

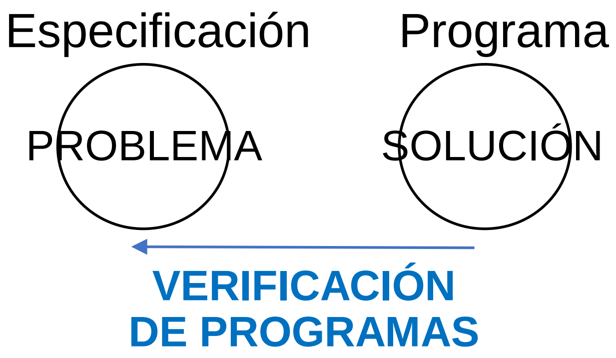
- Artefactos básicos

  Un **lenguaje de especificación** para describir los problemas (especificaciones).
  Un **lenguaje de programación** para describir las soluciones (programas).

- Planteamos una metodología para **verificar la correctitud de un programa con respecto a una especificación.** Es decir, ahora nos centramos en la **correctitud** de las soluciones.

- 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, como por ejemplo los programas concurrentes, con diversidad de procedimientos, etc).

- *Approach* a estudiar

  Axiomático, basado en la **lógica de predicados**, con **axiomas** y **reglas de inferencia** asociados a las instrucciones de los programas. Las pruebas son **sintácticas**, en lugar de manipular **estados** de variables se manipulan **predicados** como **abstracción** de los mismos.

Ejemplo de prueba axiomática en la aritmética

Prueba del teorema **1 + 1 = 2** de la aritmética.

Axiomas y Reglas de la Lógica de Predicados

K_1 : $A \to (B \to A)$

K_2 : $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

K_3 : $(\neg A \to \neg B) \to (B \to A)$

K_4 : $(\forall x)\, A(x) \to A(x|t)$, si las variables de t están libres en A

K_5 : $(\forall x)\, (A \to B) \to (A \to (\forall x)\, B)$, si x no está libre en A

K_6 a K_{10}: Axiomas de la Igualdad

Regla de Modus Ponens (MP): a partir de A y de $A \to B$ se infiere B

Regla de Generalización: de A se infiere $(\forall x)$ A

Axiomas de la Aritmética

N_1 : $(\forall x)\, \neg(s(x) = 0)$	1er axioma del sucesor
N_2 : $(\forall x)(\forall y)(x = y \to s(x) = s(y))$	2do axioma del sucesor
N_3 : $(\forall x)(x + 0 = x)$	1er axioma de la suma
N_4 : $(\forall x)(\forall y)(x + s(y) = s(x + y))$	2do axioma de la suma
N_5 : $(\forall x)\, (x \,.\, 0 = 0)$	1er axioma de la multiplicación
N_6 : $(\forall x)(\forall y)(x \,.\, s(y) = x \,.\, y + x)$	2do axioma de la multiplicación
N_7 : $P(0) \to ((\forall x)(P(x) \to P(s(x))) \to (\forall x)\, P(x))$, x libre en P	inducción

Ejemplo de prueba axiomática en la aritmética (continuación)

1.	$(\forall x)(x + 0 = x)$	axioma N_3
2.	$(\forall x)(x + 0 = x) \rightarrow 1 + 0 = 1$	axioma K_4
3.	$1 + 0 = 1$	MP entre 1 y 2
4.	$(\forall x)(\forall y)(x + s(y) = s(x + y))$	axioma N_4
5.	$(\forall x)(\forall y)(x + s(y) = s(x + y)) \rightarrow (\forall y)(1 + s(y) = s(1 + y))$	axioma K_4
6.	$(\forall y)(1 + s(y) = s(1 + y))$	MP entre 4 y 5
7.	$(\forall y)(1 + s(y) = s(1 + y)) \rightarrow 1 + s(0) = s(1 + 0)$	axioma K_4
8.	$1 + s(0) = s(1 + 0)$	MP entre 6 y 7
9.	$x = y \rightarrow s(x) = s(y)$	axioma N_2
10.	$1 + 0 = 1 \rightarrow s(1 + 0) = s(1)$	demostrado desde 9
11.	$s(1 + 0) = s(1)$	MP entre 3 y 10
12.	$(\forall x)(\forall y)(\forall z)(x = y \rightarrow (y = z \rightarrow x = z))$	teorema de la aritmética
13.	$1 + s(0) = s(1 + 0) \rightarrow (s(1 + 0) = s(1) \rightarrow 1 + s(0) = s(1))$	demostrado desde 12
14.	$s(1 + 0) = s(1) \rightarrow 1 + s(0) = s(1)$	MP entre 8 y 13
15.	$1 + s(0) = s(1)$	MP entre 11 y 14

Y como $s(0)$ se abrevia con 1 y $s(1)$ se abrevia con 2, se alcanza el teorema $1 + 1 = 2$.

La prueba es **sintáctica,** sólo se usan axiomas, reglas y teoremas demostrados previamente.

Lo mínimo que se exige de una axiomática es que sea **sensata (*sound*),** que lo que pruebe se cumpla **semánticamente**.

La propiedad inversa, deseable, de una axiomática, es que sea **completa,** que pueda probar todo lo que se cumpla semánticamente (no siempre se cumple, por ejemplo la propia aritmética es incompleta).

Ejemplo de prueba axiomática de un programa

Verificación de un programa que calcula el **factorial**:

PROGRAMA

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

PRUEBA

{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 = a! ∧ a = x}
{y = x!}

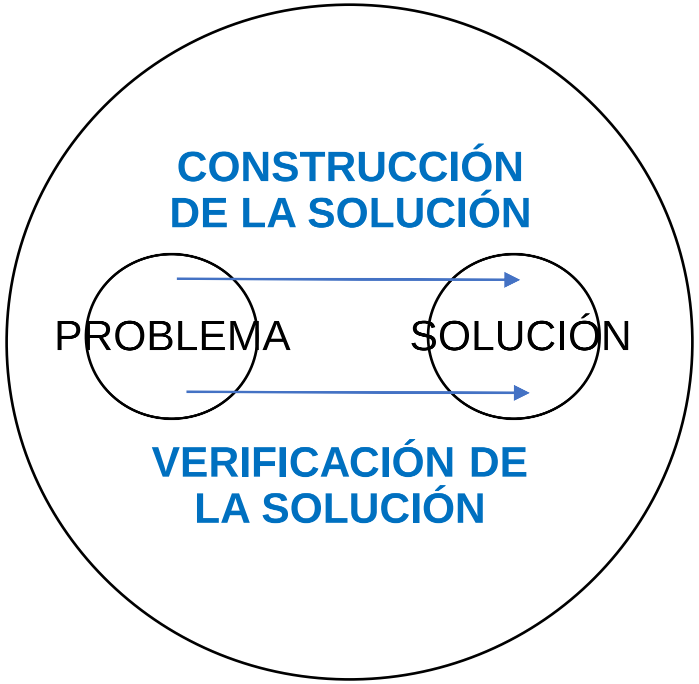
- La idea central es plantear una metodología para obtener **programas correctos por construcción**, con la guía de los axiomas y reglas definidos.

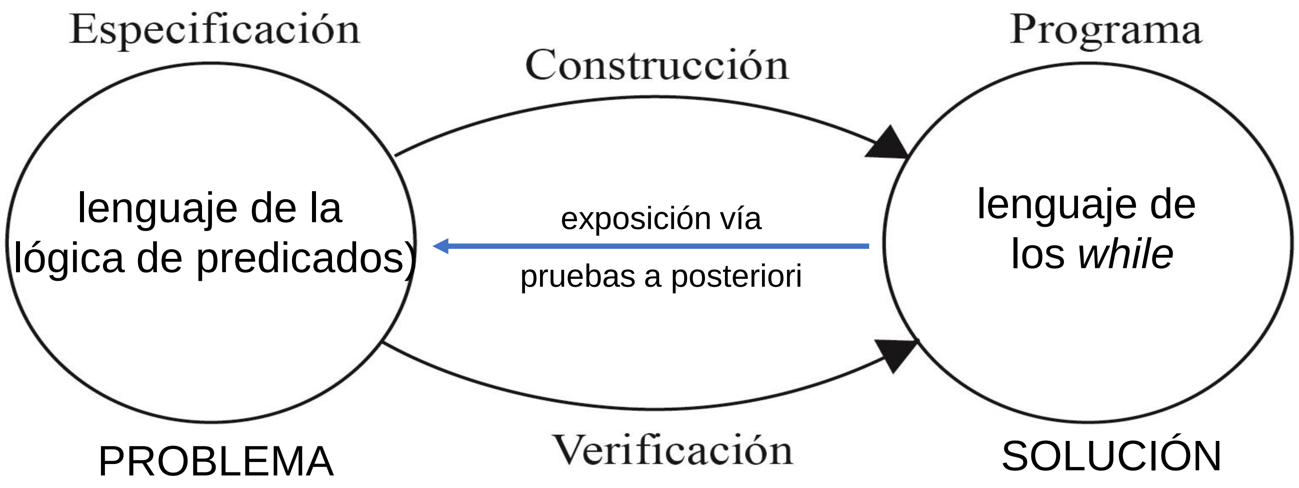
- En otras palabras, **construir y verificar programas en simultáneo**.

- Sólo por fines didácticos, la exposición del tema se hace con pruebas de programas ya construidos.

- Un programa es correcto siempre **con respecto a una especificación**.

- Utilizamos como lenguaje de especificación el de la **lógica de predicados** y como lenguaje de programación el de los *while* (**tipo Pascal**).





Lenguaje de programación

Sintaxis (en Backus-Naur Form o BNF)

- Sus instrucciones son:

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

- La expresión *e* es de tipo entero:

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

- La expresión B es de tipo booleano:

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

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

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

¿Cuál es el resultado que devuelve el programa en la variable y cuando el input x es negativo?
¿Es correcto?
¿Esto hace que el programa sea incorrecto? ¿Por qué?

Semántica (matemática o denotacional) de las expresiones

- A cada **constante** se le asigna un valor **fijo** entero o booleano. Algunos casos:
  - A la constante n, el número n.
  - A las constantes true y false, los valores verdadero y falso.
  - Al símbolo +, la función suma.
  - Al símbolo ¬, la función negación.
  - Al símbolo =, la función igualdad.

- A cada **variable** se le asigna un valor **no fijo** con una **función σ** llamada **estado.** $\sigma : Ivar \to Z$
  - σ(x) = 5, σ(y) = -1, σ(z) = 0, etc.

    Σ es el conjunto de todos los estados. Se usa σ[x|n] para indicar que la variable x tiene el valor n.

- A cada **expresión** se le asigna inductivamente, mediante una **función S**, un valor (número o valor de verdad) que depende del estado considerado:
  - S(n)(σ) = n.
  - S(true)(σ) = verdadero.
  - S(x)(σ) = σ(x).
  - $S(e_1 + e_2)(\sigma) = S(e_1)(\sigma) + S(e_2)(\sigma)$.
  - $S(e_1 = e_2)(\sigma) = (S(e_1)(\sigma) = S(e_2)(\sigma))$.
  - S(¬B)(σ) = ¬S(B)(σ)
  - Etc.

    Se usa σ(e) para abreviar S(e)(σ) y σ(B) para abreviar S(B)(σ).

$S : Iexp \to (\Sigma \to Z)$

$S : Bexp \to (\Sigma \to \{verdadero, falso\})$.

Semántica (operacional) de las instrucciones

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

  1. (skip, σ) $\rightarrow$ (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) $\rightarrow$ (E, σ[x|σ(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, σ) $\rightarrow$ (S', σ'), entonces (S ; T, σ) $\rightarrow$ (S' ; T, σ') 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 σ(B) = verdadero, entonces (if B then S_1 else S_2 fi, σ) $\rightarrow$ (S_1, σ)
     = falso, entonces (if B then S_1 else S_2 fi, σ) $\rightarrow$ (S_2, σ)
     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. Si σ(B) = verdadero, entonces (while B do S od, σ) $\rightarrow$ (S ; while B do S od, σ)
     = falso, entonces (while B do S od, σ) $\rightarrow$ (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** π(S, σ), que es la secuencia de configuraciones producida por la ejecución de S a partir de σ.
  P.ej., (x := 0 ; y := 1 ; z := 2, σ) $\rightarrow$ (y := 1 ; z := 2, σ[x|0]) $\rightarrow$ (z := 2, σ[x|0][y|1]) $\rightarrow$ (E, σ[x|0][y|1][z|2]).

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

Lenguaje de especificación

Sintaxis (en BNF)

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

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

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

Semántica (matemática o denotacional)

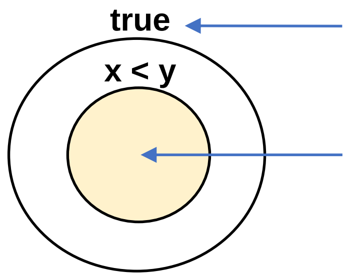
- A cada aserción se le asigna inductivamente, mediante la **función S** definida antes, un valor de verdad que depende del estado considerado.
  - $S(true)(\sigma) = verdadero$
  - $S(\neg p)(\sigma) = \neg S(p)(\sigma)$
  - $S(p \vee q)(\sigma) = S(p)(\sigma) \vee S(q)(\sigma)$
  - $S(\exists x{:}\ p)(\sigma) = verdadero$ sii $S(p)(\sigma[x|n]) = verdadero$ para algún número n
  - $S(\forall x{:}\ p)(\sigma) = verdadero$ sii $S(p)(\sigma[x|n]) = verdadero$ para todo número n
  - Etc.

$S : Aser \rightarrow (\Sigma \rightarrow \{verdadero, falso\})$

- Si $S(p)(\sigma) = verdadero$, se dice que **σ satisface p**, o que **p es verdadera cuando se evalúa en σ.**
- La notación **σ |= p** abrevia $S(p)(\sigma) = verdadero$. Lo mismo, **σ |≠ p** abrevia $S(p)(\sigma) = falso$.

  P.ej., Si $\sigma(x) = 1$ y $\sigma(y) = 2$, entonces $\sigma \models x < y$

- 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).



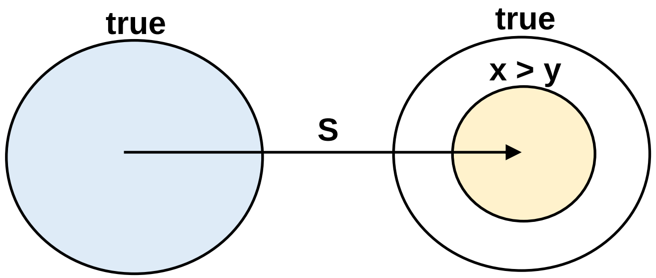
- Una **especificación** de un programa S es un par de aserciones (p, q) asociadas a la **entrada** y la **salida** de S, respectivamente. La aserción p es la **precondición** de S, denota al conjunto de estados iniciales de S. La aserción q es la **postcondición** de S, denota al conjunto de estados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

- Por ejemplo, la especificación (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).

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

Métodos axiomáticos de verificación de programas

- En los programas secuenciales se consideran básicamente dos propiedades, la **correctitud parcial** y la **terminación** (o **no divergencia),** 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:

$$(\sigma \models p \ \wedge \ val(\pi(S, \sigma)) \neq \perp) \rightarrow val(\pi(S, \sigma)) \models q$$

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

  La expresión $\models \{p\}\ S\ \{q\}$ denota que S es parcialmente correcto con respecto a (p, q).

Ejercicio: ¿Se cumple lo siguiente? (a) $\models$ {true} skip {true}. (b) $\models$ {x = 0} x := x + 1 {x = 1}. (c) $\models$ {true} x := x + 1 {x = 1}

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

$$\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \ \wedge \ val(\pi(S, \sigma)) \models q)$$

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

  La expresión $\models \langle p \rangle\ S\ \langle q \rangle$ denota que S es totalmente correcto con respecto a (p, q).
  En particular, $\models \langle p \rangle\ S\ \langle true \rangle$ denota que a partir de un estado que satisface p, S termina (en algún estado).

Ejercicio: ¿Se cumple lo siguiente? (a) $\models$ $<x \neq 0>$ while x = 0 do skip od $<x \neq 0>$. (b) $\models$ $<true>$ while x $\neq$ 0 do x := x – 1 od $<true>$

- La división entre correctitud parcial y no divergencia no es caprichosa, se debe a que las dos pruebas 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 $\models \langle p \rangle\ S\ \langle q \rangle$ en dos partes: (a) $\models \{p\}\ S\ \{q\}$, (b) $\models \langle p \rangle\ S\ \langle true \rangle$.

  - Para la parte (a) se usa el **método de prueba de correctitud parcial H**. Dicho método, compuesto por axiomas y reglas, permite probar sintácticamente la fórmula semántica $\models \{p\}\ S\ \{q\}$, lo que se expresa así: **$\vdash_H \{p\}\ S\ \{q\}$** (notación heredada de la lógica).

  - Para la parte (b) se usa el **método de prueba de no divergencia H***. Es una extensión de H. Permite probar sintácticamente la fórmula semántica $\models \langle p \rangle\ S\ \langle true \rangle$, lo que se expresa con: **$\vdash_{H^*} \langle p \rangle\ S\ \langle true \rangle$.**

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

    Si $\vdash_H \{p\}\ S\ \{q\}$, entonces $\models \{p\}\ S\ \{q\}$
    Si $\vdash_{H^*} \langle p \rangle\ S\ \langle true \rangle$, entonces $\models \langle p \rangle\ S\ \langle true \rangle$

  - También es deseable que se cumpla la propiedad inversa, que los métodos sean **completos**:

    Si $\models \{p\}\ S\ \{q\}$, entonces $\vdash_H \{p\}\ S\ \{q\}$
    Si $\models \langle p \rangle\ S\ \langle true \rangle$, entonces $\vdash_{H^*} \langle p \rangle\ S\ \langle true \rangle$

- En las próximas clases vamos a describir y dar ejemplos de los métodos H y H*. Probaremos además su sensatez y completitud.

Anexo de la clase teórica 10

Introducción a la
Verificación de Programas

Lema de la forma de las computaciones

- A partir de la definición por **inducción estructural (o comprensión)** de la semántica operacional de los programas con *while* (uso de la relación de transición $\longrightarrow$), la definición por **extensión** de dicha semántica, o lo que es lo mismo, las formas de las computaciones de cada uno de los programas, se pueden desarrollar fácilmente.

- 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. 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 (queda como ejercicio).

Clase práctica 10

Introducción a la Verificación de Programas

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:

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

- 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. Se pretende especificar un programa tal que al final se cumpla la condición y = 1 ó y = 0, según al comienzo valga o no, respectivamente, la propiedad p(x), dada una variable de programa x. Se asume la existencia en el lenguaje de programación de la instrucción de asignación x := e (asignación del valor de una expresión entera *e*, a la variable entera x), y de la instrucción de secuencia, denotada con el operador **;**.

Una primera versión de la especificación, **errónea**, sería:

$\Phi_1 = (\text{true}, (y = 1 \rightarrow p(x)) \wedge (y = 0 \rightarrow \neg p(x)))$.
Notar que el programa S :: y := 2, satisface Φ_1 pero no es el programa que se pretende especificar. Acá el error es que la postcondición es demasiado débil, en el sentido lógico.

Una segunda versión de la especificación, también **errónea**, sería:

$\Phi_2 = (\text{true}, (0 \leq y \leq 1) \wedge (y =1 \rightarrow p(x)) \wedge (y = 0 \rightarrow \neg p(x)))$.
Sea el programa S :: x := 5 ; y := 1. Notar que si al comienzo, el valor de x es 7, no se cumple p(7), y se cumple p(5), entonces el programa S satisface Φ_2 y así otra vez, no es el programa que se pretende especificar. Acá el error es que se omite que la variable x puede ser modificada.

Finalmente, el siguiente intento resulta **exitoso**:

$\Phi_3 = (x = X, (0 \leq y \leq 1) \wedge (y = 1 \rightarrow p(X)) \wedge (y = 0 \rightarrow \neg p(X)))$. El uso de la variable lógica X (también llamada de especificación) subsana el problema del intento anterior, congelando el valor inicial de x.

<u>Ejemplo 3</u>. 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 σ |= true. Si $\text{val}(\pi(S, \sigma)) = \perp$, entonces se cumple la implicación trivialmente.
Y si $\text{val}(\pi(S, \sigma)) \neq \perp$, entonces $\text{val}(\pi(S, \sigma)) = \sigma' \models \text{true}$, por lo que también en este caso se cumple la implicación.

<u>Ejemplo 4</u>. ¿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.

<u>Ejemplo 5</u>. 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:

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

Se cumple σ |= true. Si $\text{val}(\pi(S, \sigma)) \neq \perp$, entonces $\text{val}(\pi(S, \sigma)) = \sigma' \models \text{false}$ (absurdo). Por lo tanto $\text{val}(\pi(S, \sigma)) = \perp$, es decir, el programa S no termina a partir de σ.

Ejemplo 6. Lema de Separación.

Vale $|= \langle p \rangle\ S\ \langle q \rangle \leftrightarrow (|= \{p\}\ S\ \{q\} \wedge |= \langle p \rangle\ S\ \langle true \rangle)$.

- Primero se probará $|= \langle p \rangle\ S\ \langle q \rangle \rightarrow (|= \{p\}\ S\ \{q\} \wedge |= \langle p \rangle\ S\ \langle true \rangle)$:

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

  Así, por (a) y (b): $|= \{p\}\ S\ \{q\} \wedge |= \langle p \rangle\ S\ \langle true \rangle$.

- Ahora se probará $(|= \{p\}\ S\ \{q\} \wedge |= \langle p \rangle\ S\ \langle true \rangle) \rightarrow |= \langle p \rangle\ S\ \langle q \rangle$:

  Sea $|= \{p\}\ S\ \{q\} \wedge |= \langle p \rangle\ S\ \langle true \rangle$. Entonces, dado σ, vale:
  (a) $(\sigma |= p \wedge val(\pi(S, \sigma)) \neq \perp) \rightarrow val(\pi(S, \sigma)) |= q$.
  (b) $\sigma |= p \rightarrow (val(\pi(S, \sigma)) \neq \perp \wedge val(\pi(S, \sigma)) |= true)$.
  (c) Por (b): $\sigma |= p \rightarrow val(\pi(S, \sigma)) \neq \perp$.
  (d) Por (a) y (b): $\sigma |= p \rightarrow val(\pi(S, \sigma)) |= q$.

  Así, por (c) y (d): $\sigma |= p \rightarrow (val(\pi(S, \sigma)) \neq \perp \wedge val(\pi(S, \sigma)) |= q)$, es decir $|= \langle p \rangle\ S\ \langle q \rangle$.