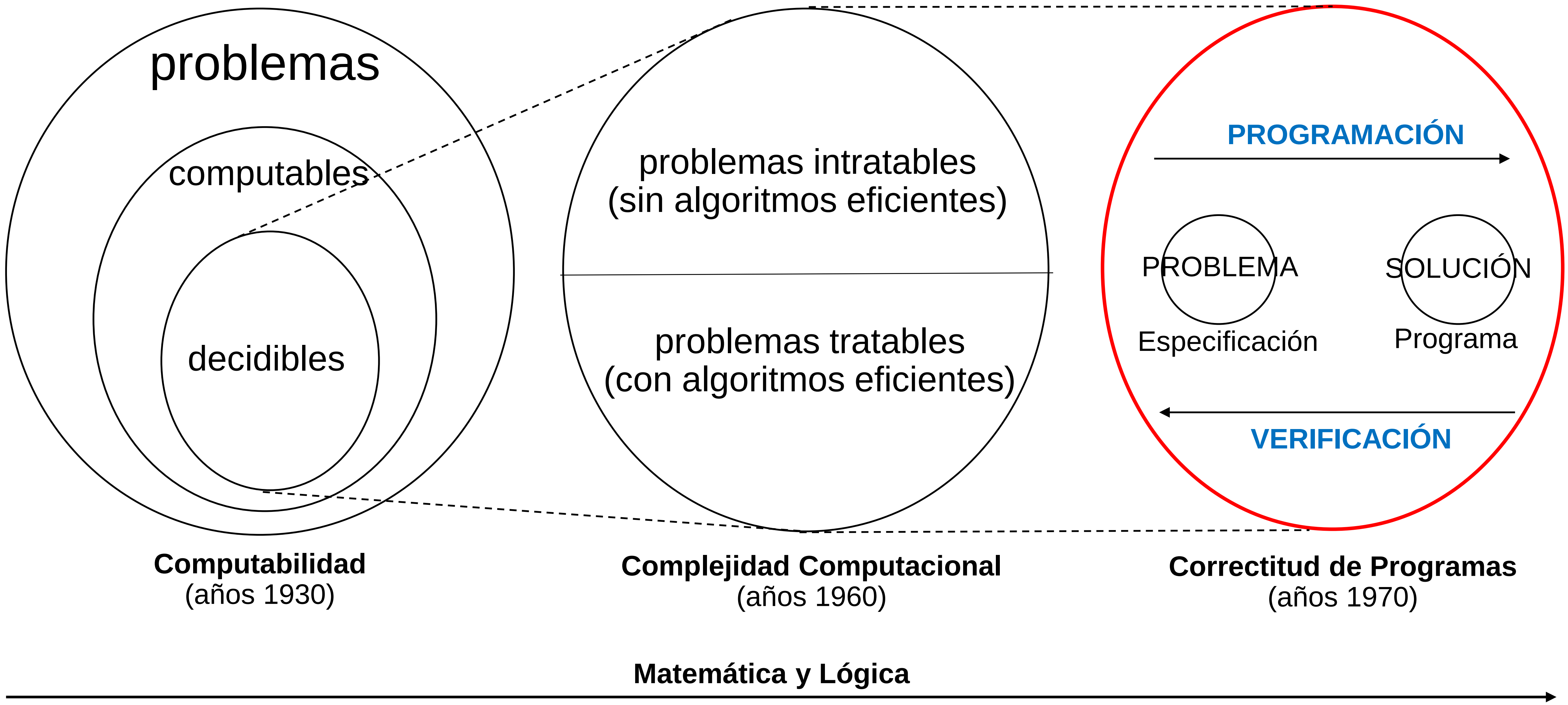
Clase teórica 10

Verificación axiomática de programas

Seguimos dentro de los problemas decidibles



Bibliografía

Libro de cabecera (en IDEAS):

R. Rosenfeld, 2024. *Verificación de programas. Programas secuenciales y concurrentes*. EDULP.

Otros libros (en biblioteca o en IDEAS):

N. Francez, 1992. *Program Verification*. Addison-Wesley.
K. Apt y F. Olderog, 1997. *Verification of Sequential and Concurrent Programs*. Springer.
M. Huth y M. Ryan, 2004. *Logic in Computer Science*. Cambridge University Press.
R. Rosenfeld y J. Irazábal, 2013. Computabilidad, Complejidad Computacional y Verificación de Programas. EDULP.
R. Rosenfeld y J. Irazábal, 2010. Teoría de la Computación y Verificación de Programas. McGraw Hill y EDULP.
C. Pons, R. Rosenfeld y C. Smith, 2017. *Lógica para Informática*. EDULP.

Algunos artículos a presentar (en IDEAS):

C. Hoare, 1969. *An axiomatic basis for computer programming*. Communications of the ACM.
K. Apt, 1981. *Ten years of Hoare´s logic*. ACM Transactions on Programming Languages and Systems.
K. Apt y E. Olderog, 2019. *Fifty years of Hoare's logic*. Formal Aspects of Computing.
E. Clarke, J. Wing et al, 1996. *Formal Methods: State of the Art and Future Directions*. ACM Computing Surveys.

Introducción. Conceptos básicos.

- ¿Cómo probar un programa?

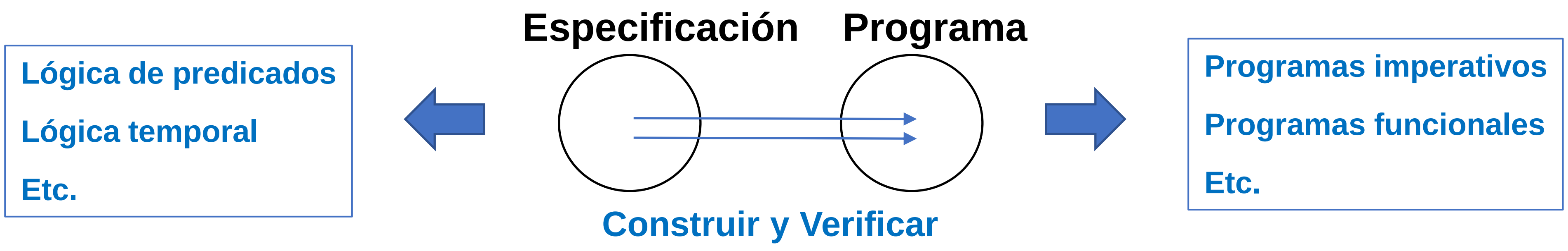
- **Verificación** (actividad *formal*) vs **validación** (actividad *informal*, fundamentalmente el *testing*).

- Dijkstra: “El testing asegura la presencia de errores pero no su ausencia”.

- Hoare: “Todas las propiedades de un programa pueden probarse en principio a partir de su propio texto por medio del puro razonamiento deductivo”.

- ¿Programar y después verificar? ¿O programar y verificar simultáneamente?

- Dijkstra: “Pensar cómo sería la prueba de un programa, y luego construirlo siguiendo la estructura de la prueba, para así construirlo y probarlo al mismo tiempo y obtener un programa correcto por construcción”.



- ¿Cómo verificar un programa? (sólo por fines didácticos asumiremos programas ya construidos)

Ejemplo 1. Verificar el siguiente programa S_{swap} que permuta x con y:

Ejemplo

S_{swap}::	$(x = 1, y = 2)$
$z := x$;	$z := 1$;
$x := y$;	$x := 2$;
$y := z$	$y := 1$

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

Dos maneras para hacerlo son:

1) Por la vía **semántica**, usando la semántica de las instrucciones del lenguaje.

	variables	*programa*
1)	$x = X, y = Y$	$\mathbf{z := x}$; $x := y$; $y := z$
2)	$x = X, y = Y,\ \mathbf{z = X}$	$\mathbf{x := y}$; $y := z$
3)	$\mathbf{x = Y}, y = Y,\ z = X$	$\mathbf{y := z}$
4)	$x = Y,\ \mathbf{y = X}, z = X$	

2) Por la vía **sintáctica**, usando axiomas y reglas de un método deductivo.

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

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

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

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

La verificación semántica se complica mucho cuando los programas son complejos (sobre todo concurrentes). Avanzaremos en lo que sigue con la verificación sintáctica (o axiomática), conocida como **Lógica de Hoare**.

Ejemplo 2. Prueba axiomática (de la aritmética)

Prueba de 1 + 1 = 2

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

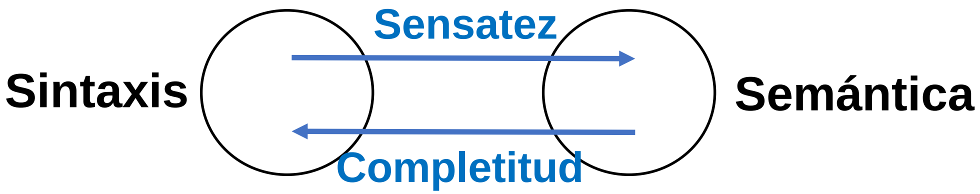
Axiomas de la Aritmética

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 : $(\forall x)\ (x \cdot 0 = 0)$	1er axioma de la multiplicación
N_6 : $(\forall x)(\forall y)(x \cdot s(y) = x \cdot y + x)$	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

Prueba

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
16.	$1 + 1 = 2$	Abreviación de 15

- Lo básico que se exige de un método axiomático es que sea **sensato**: que no pruebe enunciados falsos.
- Es ideal que también sea **completo**: que pruebe todos los enunciados verdaderos.

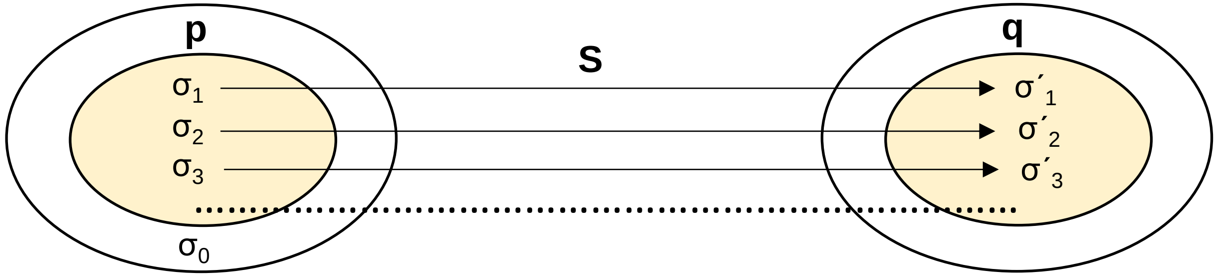


Algunas definiciones

- Volviendo al ejemplo del programa de swap:
  - $\{x = X \wedge y = Y\}\ S_{swap}\ \{y = X \wedge x = Y\}$ es una **terna de Hoare** o **fórmula de correctitud**.
  - El predicado $x = X \wedge y = Y$ es la **precondición** de S_{swap}.
  - El predicado $y = X \wedge x = Y$ es la **postcondición** de S_{swap}.
  - El par $(x = X \wedge y = Y,\ y = X \wedge x = Y)$ es la **especificación** de S_{swap}.
  - Un **estado** σ es una función que asigna a toda variable un valor. Por ejemplo: $\sigma(x) = 1$, $\sigma(y) = 2$, etc.
  - Un estado σ **satisface** un predicado p, si p evaluado con σ es verdadero. Se expresa así: **σ |= p**.
    Por ejemplo: si σ(x) = 1 y σ(y) = 2, entonces σ |= x < y.

- Un programa S es **correcto con respecto a una especificación** (p, q), lo que se expresa con **{p} S {q}**, sii:

Para todo estado σ, **si σ |= p entonces S ejecutado a partir de σ termina en un estado σ´ tal que σ´ |= q**



P. ej., si **p = (x = X > 0)** y **q = (y = 2.X),** S debe hacer:
Si σ_1 |= x = 1, entonces σ'_1 |= y = 2.
Si σ_2 |= x = 2, entonces σ'_2 |= y = 4.
Si σ_3 |= x = 3, entonces σ'_3 |= y = 6.
Etc.

¿Se cumple {p} S {q} si desde σ_0 , S no alcanza un σ'_0 dentro de q? Sí: si σ |≠ p, {p} S {q} es trivialmente verdadera.

Ejemplo 3. Verificar axiomáticamente un programa S_{fac} que calcula el factorial:

- Definición: el factorial x! de un número x > 0 es: x! = 1.2.3…x. Por ejemplo, 4! = 1.2.3.4 = 24.

- Sea:

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

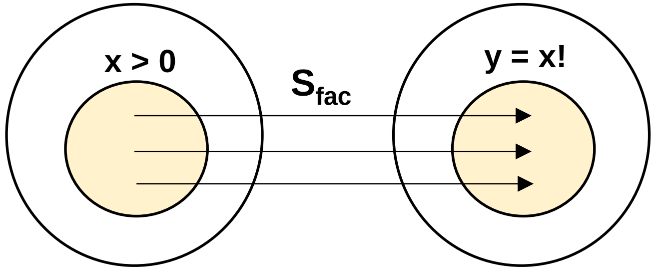
Ejemplo

(x = 4)
a = 1, y = 1
1 < 4: a = 2, y = 1.2 = 2
2 < 4: a = 3, y = 2.3 = 6
3 < 4: a = 4, **y = 6.4 = 24**

Se quiere verificar: **{x > 0} S_{fac} {y = x!}**

- Es decir, hay que probar que: **desde todo estado σ |= x > 0, S_{fac} termina en un estado σ´ |= y = x!**

- Gráficamente:



Si x = 1, luego de S_{fac} debe valer y = 1.

Si x = 2, luego de S_{fac} debe valer y = 2.

Si x = 3, luego de S_{fac} debe valer y = 6.

Etc.

Notar que si x ≤ 0, queda y = 1 ≠ x! (no es correcto). ¿Esto significa que el programa es incorrecto?

NO. La precondición considera x > 0.

Componentes del método de verificación axiomática (Lógica de Hoare)

1. Lenguaje de programación (programas con while):

- Instrucciones:

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

- Expresiones de tipo entero:

  e :: n | x | (e_1 + e_2) | (e_1 – e_2) | (e_1 . e_2) | …

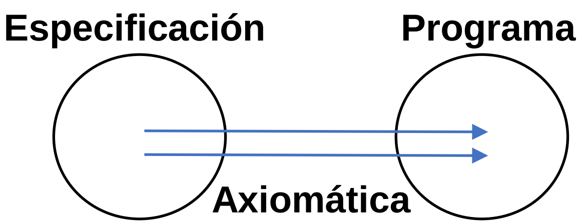
  n es una constante entera. x es una variable entera.

- Expresiones de tipo booleano:

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

2. Lenguaje de especificación (lógica de predicados):

p :: true | false | (e_1 = e_2) | (e_1 < e_2) | … | ¬p | ($p_1 \vee p_2$) | … | ∃x: p | ∀x: p



Ejemplo de programa

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

Ejemplos de predicados

true
x + 1 = y
¬(a < x)
∀x: (x > y ∨ x ≤ y)

Para simplificar, si se puede se suprimen los paréntesis

3. Axiomática:

1. Axioma de la asignación (ASI)

$\{p(e)\}\ x := e\ \{p(x)\}$

Si luego de x := e vale p para x, entonces antes de x := e valía p para *e*.

Por ejemplo: $\{y > 0\}\ x := y\ \{x > 0\}$

Ejercicio: $\{?\}\ x := x + 1\ \{x > 0]$

2. Regla de la secuencia (SEC)

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

3. Regla del condicional (COND)

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

4. Regla de la repetición (REP)

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

$\{p\}$
while B do
$\{p \wedge B\}$ t
S t
$\{p\}$
od ....
$\{p \wedge \neg B\}$ t

p y t se definen en términos de las variables del programa S.
p es un predicado que vale antes y después de toda iteración (**invariante**).
t es una función entera que decrece después de toda iteración (**variante**).

¿Para qué se necesita la condición $p \rightarrow t \geq 0$? Si t “se pasa” a los negativos hay infinitud.

Ejemplo 4. Verificación de un programa S_{abs} que calcula el valor absoluto de un número entero.

S_{abs}::
if x > 0 then y := x
        else y := – x
fi

Hay que verificar: **{x = X} S_{abs} {y = |X|}**

Axioma de la asignación (ASI)

{p(e)} x := e {p(x)}

Regla del condicional (COND)

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

Prueba

1. {x = |X|} y := x {y = |X|}	por el axioma ASI
2. {–x = |X|} y := –x {y = |X|}	por el axioma ASI
3. {x = X ∧ x > 0} y := x {y = |X|}	de (1), porque (x = X ∧ x > 0) implica x = |X|
4. {x = X ∧ ¬(x > 0)} y := – x {y = |X|}	de (2), porque (x = X ∧ ¬(x > 0)) implica –x = |X|
5. {x = X} if x > 0 then y := x else y := – x fi {y = |X|}	de (3) y (4), por la regla COND

Otra manera de presentar la prueba (*proof outline* o *esquema de prueba*):

{x = X}
if x > 0 then {x = X ∧ x > 0} y := x {y = |X|}
        else {x = X ∧ ¬(x > 0)} y := – x {y = |X|}
fi
{y = |X|}

Ejemplo 5. Verificación del programa S_{fac} mostrado previamente (ejemplo 3).

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

Hay que verificar: **{x > 0} S_{fac} {y = x!}**

Regla de la Repetición (REP)

1) $\{p \wedge B\}\ S\ \{p\}$
2) $\{p \wedge B \wedge t = Z\}\ S\ \{t < Z\}$
3) $p \rightarrow t \geq 0$

---

4) $\{p\}$ while B do S od $\{p \wedge \neg B\}$

Fragmento inicial: **{x > 0}** **a := 1 ; y := 1** $\underbrace{\{y = a! \wedge a \leq x\}}_{\textbf{invariante}}$

Repetición:

1) $\{(y = a! \wedge a \leq x) \wedge (a < x)\}$
   a := a + 1 ;
   y := y . a
   $\{y = a! \wedge a \leq x\}$

2) $\{(y = a! \wedge a \leq x) \wedge (a < x) \wedge \underbrace{(x - a = Z)}_{\textbf{variante}}\}$
   a := a + 1 ;
   y := y . a
   $\{x - a < Z\}$

3) $(y = a! \wedge a \leq x) \rightarrow (x - a \geq 0)$

4) $\{y = a! \wedge a \leq x\}$
   while a < x do a := a + 1 ; y := y . a od
   $\{(y = a! \wedge a \leq x) \wedge \neg(a < x)\}$, equivalente a : **{y = x!}**

Secuencia: **{x > 0}** S_{fac} **{y = x!}**

Proof Outline

{x > 0}
a := 1 ; y := 1 ;
{inv: p = $(y = a! \wedge a \leq x)$, var: t = (x – a)}
while a < x do
$\{p \wedge (a < x)\}$
a := a + 1 ; y := y . a
{p}
od
$\{p \wedge \neg(a < x)\}$
{y = x!}

t = x – 1 → **t = x – 2** → .... **t = 0**

¿Qué representa el valor de t? El tope de iteraciones

Composicionalidad

- El método de prueba presentado es **composicional**:
  Dado un programa S, compuesto por subprogramas $S_1, \ldots, S_n$,
  que valga la fórmula {p} S {q} depende **sólo** de que valgan 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**).

- Por ejemplo, dado el programa S :: S_1 ; S_2, si se cumplen las fórmulas: **$\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$,**
  también se cumple la fórmula: **$\{p\}\ S_1 ; S_2\ \{q\}$,**
  independientemente del contenido de S_1 y S_2.

- Más aún, si en lugar de **S_2** utilizamos un subprograma **S_3** que también satisface la fórmula: **$\{r\}\ S_3\ \{q\}$,**
  entonces también se cumple la fórmula: **$\{p\}\ S_1 ; S_3\ \{q\}$,**
  lo que significa que S_2 y S_3 son **intercambiables** (son funcionalmente equivalentes respecto de (r, q)).

$$\{p\}\ \boxed{S_1}\ \{r\}\ \boxed{S_2}\ \{q\} \qquad\qquad \{p\}\ \boxed{S_1}\ \{r\}\ \boxed{S_3}\ \{q\}$$

- **En los programas concurrentes la composicionalidad se pierde:**

Especificaciones

- Hemos especificado un programa para calcular el factorial de la siguiente forma:

$$(x > 0,\ y = x!)$$

¿Pero es correcta la especificación?

No. Por ejemplo, el programa **S :: x := 1 ; y := 1** satisface (x > 0, y = x!) pero no es el programa pedido: Por ejemplo, si al inicio x = 5, entonces al final debe ser y = 5! = 120.

- Lo que sucede es que **las variables de la precondición pueden modificarse a lo largo del programa**.

- Lo que se hace es utilizar **variables lógicas**, para **congelar valores**. En ej ejemplo considerado haríamos:

$$(x = X \wedge X > 0,\ y = X!\}$$

¿Y se puede agregar a la especificación que la variable x no se modifique nunca?

No, la lógica de predicados no lo permite. Una lógica que sí lo permite es la **lógica temporal**.

Sensatez, completitud y automatización de la verificación

Sensatez

- Si se cumple sintácticamente {p} S {q} (con axiomas y reglas),
  ¿se cumple semánticamente {p} S {q} (considerando estados y computaciones)?
  Propiedad **obligatoria**.

Completitud

- Si se cumple semánticamente {p} S {q} (con estados y computaciones),
  ¿se puede probar sintácticamente {p} S {q} (considerando axiomas y reglas)?
  Propiedad **deseable**.
- Su cumplimiento depende de la **expresividad** del lenguaje de especificación. Por ejemplo:
  Dada la fórmula {p} S_1 ; S_2 {q}, debe poder encontrarse **un predicado intermedio entre S_1 y S_2.**
  Dada la fórmula {r} while B do S od {q}, debe poder encontrarse **un invariante para el while.**

Automatización

- La verificación de programas es en general **indecidible**, y por lo tanto **no automatizable**.
- Existen numerosas **herramientas** que asisten interactivamente al programador.
- En particular, si los programas son de estados finitos, existen sistemas de verificación automática, los sistemas de **model checking** (utilizan lógica temporal, y no son sintácticamente orientados como la Lógica de Hoare).

Anexo

Soporte herramental (Lógica de Hoare y otros formalismos)

- Uso industrial y académico.

- **Verificación axiomática**.

  Entornos interactivos de asistencia al programador, con compilación, deducción, manipulación lógica y aritmética, etc. Algunos ejemplos:

  - **Dafny**. Basado en la lógica de Hoare. Lenguaje compilado enfocado en C#.
  - **COQ** (INRIA). Basado en la teoría de tipos.
  - **Isabelle**. Framework con un lenguaje lógico fuertemente tipado.

- **Model checking (no basado en la Lógica de Hoare).**

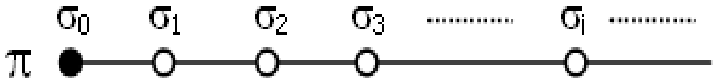
  Verificación automática basada en especificaciones con lógica temporal lineal o computacional. Algunos ejemplos:

  - **EMC** y **CAESAR** (Emerson & Clarke, Queille y Sifakis) fueron los primeros model checkers.
  - **SMV**. Verificación de modelos basados en BDDs (Binary Decision Diagrams) y lógica temporal CTL.
  - **SPIN**. Centrado en el problema de la explosión de estados. Lenguaje temporal LTL.

- **Síntesis de programas.** Enfoque reciente con **programación por ejemplos (búsqueda estocástica, IA).**

Acerca de la lógica temporal

- La **lógica temporal** permite especificar propiedades **a lo largo de computaciones.**



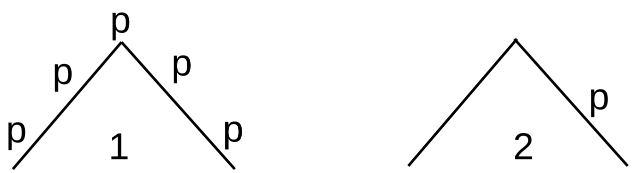
Por ejemplo, permite establecer que a partir de σ_0 **siempre** vale $x > 0$.

- En esencia, extiende la lógica de predicados con **operadores temporales**. Ejemplo de fórmulas:

1. $\sigma_0 \models Xp$ significa que en el estado siguiente de σ_0 vale p.
2. $\sigma_0 \models Gp$ significa que a partir de σ_0 siempre vale p.
3. $\sigma_0 \models Fp$ significa que en algún estado siguiente de σ_0 vale p.
4. $\sigma_0 \models p\ U\ q$ significa que a partir de σ_0 vale repetidamente p y en algún momento vale q (p puede o no seguir valiendo).

- Hay dos familias de lenguajes de la lógica temporal, **LTL** (lógica lineal), definidos sobre computaciones (como la que mostramos), y **CTL** (lógica computacional o arbórea), definidos sobre **árboles de computaciones**, lo que permite especificar computaciones específicas. Ejemplo de fórmulas CTL:

1. $\sigma_0 \models AGp$ significa que sobre toda computación desde σ_0 se cumple Gp
2. $\sigma_0 \models EFp$ significa que sobre alguna computación desde σ_0 se cumple Fp



Ninguna familia es más expresiva que la otra. Hay controversias sobre la conveniencia del uso de una sobre la otra. Existen algoritmos de verificación basados en ambos tipos de lenguajes.

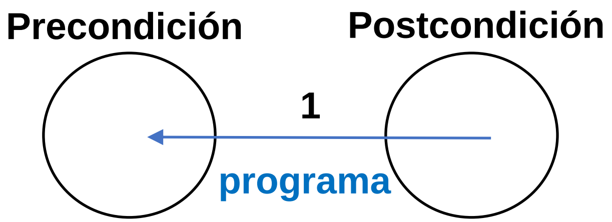
Algunas extensiones y aplicaciones de la Lógica de Hoare

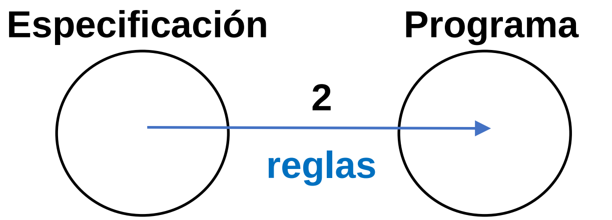
EXTENSIONES

- **Procedimientos.** Distintos tipos de pasajes de parámetros y recursión.
- **Datos.** Estructuras de datos, variables locales, punteros.
- **Programas concurrentes.** Programas paralelos (memoria compartida) y distribuidos (pasajes de mensajes).
- **Programas probabilísticos y cuánticos.**

APLICACIONES (CÁLCULO DE PROGRAMAS)

1. **Precondición más débil** (conjunto de estados iniciales más amplio posible)
2. **Síntesis de programas** (transformaciones a partir de la especificación)
3. Otros métodos de cálculo de programas





Verificación de programas con procedimientos

- Caso **no recursivo y sin parámetros**:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \text{call proc}\ \{q\}}$$

siendo S el cuerpo del procedimiento proc (S es la macro expansión de proc).

- Caso **no recursivo y con parámetros**. P.ej. con pasajes por valor y resultado (uso de la sustitución lógica):

$$\frac{p \rightarrow p'[y|e],\ \{p'\}\ S(y,\ x)\ \{q'\},\ q'[x|v] \rightarrow q}{\{p\}\ \text{call proc}\ (e,\ v)\ \{q\}}$$

siendo los parámetros reales la expresión *e* pasada por valor y la variable v pasada por resultado. Los parámetros formales son las variables *y* y x (x tiene el resultado).

- Caso **recursivo y sin parámetros**:

$$\frac{\{p\}\ \text{call proc}\ \{q\}\ |\!-\ \{p\}\ S\ \{q\}}{\{p\}\ \text{call proc}\ \{q\}}$$

{p} call proc {q}

{p}
proc
- - - - - - -
{p}
call proc
{q}
- - - - - - -
end
{q}

S

Si con la hipótesis {p} call proc {q} se prueba {p} S {q}, entonces vale {p} call proc {q}.
El call proc de arriba es interno a S, y el call proc de abajo es el que invoca a S.

Verificación con estructuras de datos (en general y con concurrencia)

Estructuras de datos (en general)

- Uso de **tipos de datos abstractos** como método de programación ampliamente aceptado (Hoare).
- Idea: **postergar la representación de los datos** hasta la instancia apropiada.
  1. Programa con tipos de datos abstractos.
  2. **Verificación del programa abstracto.**
  3. Representación de los tipos de datos abstractos.
  4. **Verificación de la representación.**

  La secuencia (1) y (2) puede tener varias iteraciones (varios niveles de abstracción).

Estructuras de datos (con concurrencia)

- **Recursos de variables compartidas**.
  Conjuntos disjuntos de variables, de acceso exclusivo por parte de los procesos.
  Un invariante por recurso, que vale al inicio y al final del uso del recurso.
- **Monitores y objetos**.
  Conjuntos disjuntos de variables y operaciones, de acceso exclusivo por parte de los procesos.
  Un invariante por monitor u objeto, que vale al inicio y al final del uso del monitor u objeto.

Programar y verificar en simultáneo

- Lo correcto es **programar al tiempo que verificar**, para obtener un programa **correcto por construcción.**

- El método de prueba definido es un buen soporte para dicha práctica.

- Por ejemplo, supongamos que se quiere construir un programa con la siguiente estructura:

  T ; while B do S od

  que debe ser correcto con respecto a una especificación (r, q), o sea que debe satisfacer la fórmula:

  {r} T ; while B do S od {q}

- Hay que construir el fragmento inicial T y el *while*. De acuerdo al método, se tiene que encontrar un predicado p (**invariante**) y una función t (**variante**) para el *while*, y se deben cumplir **cinco condiciones**:

1. A partir de la precondición r, el fragmento T termina estableciendo el predicado p: **{r} T {p}**
2. El predicado p es un invariante del *while*: **{p $\wedge$ B} S {p}**
3. La función t decrece después de cada iteración del *while*: **{p $\wedge$ B $\wedge$ t = Z} S {t < Z}**
4. El predicado p asegura que la función t siempre es positiva: **p $\rightarrow$ t $\geq$ 0**

   (cumplidos (2), (3) y (4), se obtiene **{p} while B do S od {p $\wedge$ ¬B}**)
5. El *while* termina estableciendo la postcondición q: **(p $\wedge$ ¬B) $\rightarrow$ q**

{r}
T ;
{p}
while B do
{p $\wedge$ B}
S
{p}
od
{p $\wedge$ ¬B}
{q}

Pérdida de la composicionalidad en los programas concurrentes

Ejemplo

- Se cumple: $\{x = 0\}\ S_1 :: x := x + 2\ \{x = 2\}$ y $\{x = 0\}\ S_2 :: z := x\ \{z = 0\}$ **pero no se cumple:** $\{x = 0 \wedge x = 0\}\ [S_1 :: x := x + 2 \parallel S_2 :: z := x]\ \{x = 2 \wedge z = 0\}$

  porque si en el programa $[S_1 \parallel S_2]$ se ejecuta S_2 después de S_1, al final se cumple $z = 2$. En efecto vale:

$$\{x = 0 \wedge x = 0\}$$
$$[S_1 :: x := x + 2 \parallel S_2 :: z := x]$$
$$\{x = 2 \wedge (z = 0 \vee z = 2)\}$$

- Notar además que cambiando $S_1 :: x := x + 2$ por el proceso equivalente $S_3 :: x := x + 1 ; x := x + 1$, no vale:

$$\{x = 0 \wedge x = 0\}$$
$$[S_3 :: x := x + 1 ; x := x + 1 \parallel S_2 :: z := x]$$
$$\{x = 2 \wedge (z = 0 \vee z = 2)\}$$

  porque si S_2 se ejecuta entre las dos asignaciones de S_3, al final se cumple $z = 1$. En efecto, vale:

$$\{x = 0 \wedge x = 0\}$$
$$[S_3 :: x := x + 1 ; x := x + 1 \parallel S_2 :: z := x]$$
$$\{x = 2 \wedge (z = 0 \vee z = 1 \vee z = 2)\}$$

  y así en la concurrencia dos procesos funcionalmente equivalentes **no son intercambiables.**

- **Se pierde la noción de caja negra**. Se plantean distintas técnicas de remediación.

Más sobre la verificación de programas concurrentes

- Verificación de **más de una computación** (modelo de *interleaving*).
- **Más propiedades para probar**: ausencia de deadlock, exclusión mutua, ausencia de inanición.
- Distintos **modelos de comunicación**: variables compartidas, pasajes de mensajes.
- **Incompletitud**. Necesidad de uso de **variables auxiliares** en los predicados.
- Distintas hipótesis de progreso de las computaciones. **Fairness**.
- Pérdida de la **composicionalidad** (formalización de lo ya dicho):

Dado $[S_1 \parallel S_2 \parallel \dots \parallel S_n]$, la regla natural de prueba es la siguiente:

$$\frac{\{p_1\}\ S_1\ \{q_1\}\ ,\ \{p_2\}\ S_2\ \{q_2\}\ ,\ \dots,\ \{p_n\}\ S_n\ \{q_n\}}{\{p_1 \land p_2 \land \dots \land p_n\}\ [S_1 \parallel S_2 \parallel \dots \parallel S_n]\ \{q_1 \land q_2 \land \dots \land q_n\}}$$

Pero esta regla **no es sensata**. Sucede que la postcondición de una instrucción de un proceso no depende solamente de las instrucciones precedentes sino de instrucciones **de otros procesos.**

$\mathbf{S_1}$		$\mathbf{S_2}$		$\mathbf{S_3}$
		$\{p_1 \land p_2 \land p_3\}$		
$\{p_1\}$		$\{p_2\}$		$\{p_3\}$
$[S_{11}$		S_{21}		S_{31}
$\{p_{12}\}$	$\parallel$	$\{p_{22}\}$	$\parallel$	$\{p_{32}\}$
S_{12}		S_{22}		S_{32}
…..		…..		…..
$\{p_{1k1}\}$		$\{p_{2k2}\}$		$\{p_{3k3}\}$
S_{1k1}		S_{2k2}		$S_{3k3}]$
$\{q_1\}$		$\{q_2\}$		$\{q_3\}$

¿$\{q_1 \land q_2 \land q_3\}$?

Sólo si los p_i son invariantes

Clase práctica 10

Ejemplo 1. Indicar y justificar semánticamente si se cumplen las siguientes ternas de Hoare.
Nota: el predicado true denota el conjunto de todos los estados.

1. {x > 0} while x ≠ 0 do x := x – 1 od {x = 0} SI. El while termina con x = 0 porque arranca con x > 0 y cada vez se resta 1 a x.
2. {true} while x ≠ 0 do x := x – 1 od {true} NO. Si el while arranca con x < 0 no termina porque cada vez x se decrementa.

Ejemplo 2. Asumiendo {p} S {q}, indicar y justificar semánticamente si se cumplen las siguientes afirmaciones:

1. Si S terminó en un estado final que no satisface q, entonces empezó en un estado inicial que no satisface p.
   Se cumple por definición, aplicando el contrarrecíproco.
2. Si S terminó en un estado final que satisface q, entonces empezó en un estado inicial que satisface p.
   No se cumple: {p} S {q} es verdadera trivialmente cuando el estado inicial no satisface p (implicación con antecedente falso).

Ejemplo 3. Aplicar el axioma de asignación (ASI) para obtener las precondiciones correspondientes:

1. {?} x := x + 1 {x + 1 ≠ 0} (x + 1) + 1 ≠ 0
2. {?} x := y {x = y} y = y

Ejemplo 4. Especificar un programa que duplique el valor de su variable de entrada.

Se debe utilizar una variable lógica: (x = X ∧ X > 0, y = 2.X).

Ejemplo 5. 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.

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.

Ejemplo 6. Se quiere probar:

$\{x \geq 0 \wedge y \geq 0 \wedge prod = 0 \wedge tope = 0\}$
while tope ≠ y do
    prod := prod + x ;
    tope := tope + 1
od
$\{prod = x.y\}$

El programa pretende obtener en *prod* el producto de x e *y*. Para su verificación puede servir el invariante p = (**prod = x.tope**) y el variante t = (**y – tope**). Comprobar informalmente que p y t cumplen con las definiciones de invariante y variante:

Invariante p

p se cumple antes del *while*:
$(x \geq 0 \wedge y \geq 0 \wedge prod = 0 \wedge tope = 0) \longrightarrow (prod = x.tope)$

p se cumple después de toda iteración:
Si p = (prod = x.tope), luego de prod := prod + x; tope := tope + 1 queda prod + x = x.(tope + 1), equivalente a prod = x.tope

Variante t

t se decrementa después de toda iteración:
Dado t = (y – tope), después de prod := prod + x ; tope := tope + 1 se llega a t´ = y – (tope + 1) < t, siendo tope > 0

t siempre es mayor o igual que cero:
t empieza con y – tope = y – 0 ≥ 0, luego de una iteración decrece y nunca es negativo porque cuando tope = y, t vale 0

Ejercicio. ¿Cómo se obtiene la postcondición prod = x.y?   Respuesta: $(prod = x.tope \wedge \neg(tope \neq y)) \longrightarrow prod = x.y$