Bibliografía

Básica

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

- Teoría de la Computación y Verificación de Programas. Rosenfeld & Irazábal. McGraw Hill y EDULP. 2010. Libro físico (en Biblioteca). *Para las materias básica y avanzada.*

- Lógica para Informática. Pons, Rosenfeld & Smith. EDULP. 2017. http://sedici.unlp.edu.ar/handle/10915/61426. *Para las materias básica y avanzada.*

Complementaria (algunos libros relevantes, todos en Biblioteca)

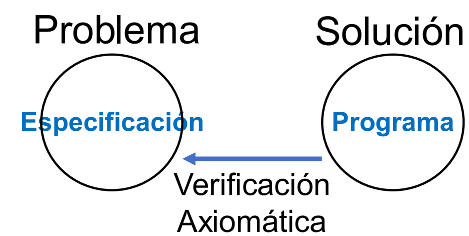
- Program Verification. Nissim Francez. Addison-Wesley. 1992.

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

- Logic in Computer Science. M. Huth y M. Ryan. Cambridge University Press. 2004.

Introducción

- Para la verificación de programas, utilizaremos los siguientes artefactos:
  - Un **lenguaje de especificación** para describir los problemas.
  - Un **lenguaje de programación** para describir las soluciones.
  - Un **método de verificación de programas** con axiomas y reglas para verificar un programa con respecto a una especificación.



- Marco de estudio (introductorio):
  - Especificaciones: con la **lógica de predicados**.
  - Programas: con un lenguaje **secuencial imperativo** (variables e instrucciones que las transforman).

- *Approachs* para verificar programas:
  - **Semántico (u operacional):** se analiza cómo una computación transforma las variables, desde el inicio hasta el final. Prohibitivo cuando se tratan programas muy complejos, en especial concurrentes.
  - **Sintáctico (o axiomático):** las pruebas se desarrollan utilizando axiomas y reglas de inferencia, cada uno asociado a una instrucción del lenguaje de programación. **Analizaremos este *approach*.**
  - **Autómatico (o *model checking*):** cuando los programas se pueden modelizar apropiadamente, la verificación puede automatizarse (con autómatas finitos o algoritmos sobre grafos, y lógica temporal).

Ejemplo de prueba axiomática en la aritmética

Prueba de $1 + 1 = 2$

Axiomas y reglas a utilizar:

Axiomas y Reglas de la Lógica de Predicados

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

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

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

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

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

Axiomas de la Igualdad (K_6 a K_{10})

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

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

Axiomas de la Aritmética

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

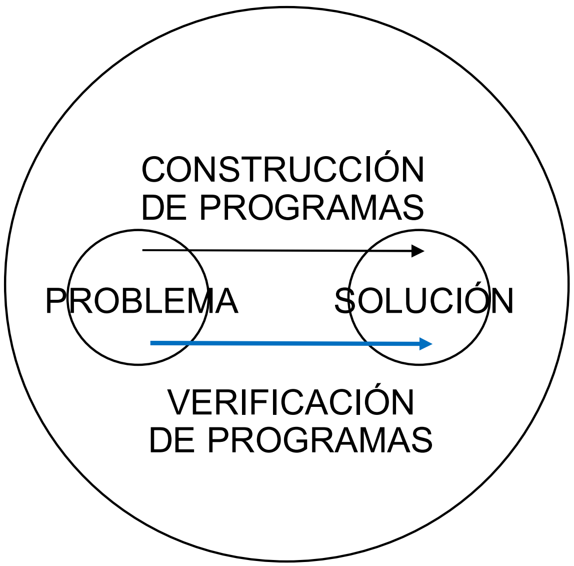
Desarrollo de la prueba:

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.	$(\forall x)(\forall y)(x + f(y) = f(x + y))$	axioma N_4
5.	$(\forall x)(\forall y)(x + f(y) = f(x + y)) \rightarrow (\forall y)(1 + f(y) = f(1 + y))$	axioma K_4
6.	$(\forall y)(1 + f(y) = f(1 + y))$	MP entre 4 y 5
7.	$(\forall y)(1 + f(y) = f(1 + y)) \rightarrow 1 + f(0) = f(1 + 0)$	axioma K_4
8.	$1 + f(0) = f(1 + 0)$	MP entre 6 y 7
9.	$x = y \rightarrow f(x) = f(y)$	axioma N_2
10.	$1 + 0 = 1 \rightarrow f(1 + 0) = f(1)$	demostrado (abrev.) desde 9
11.	$f(1 + 0) = f(1)$	MP entre 3 y 10
12.	$(\forall x)(\forall y)(\forall z)(x = y \rightarrow (y = z \rightarrow x = z))$	teorema
13.	$1 + f(0) = f(1 + 0) \rightarrow (f(1 + 0) = f(1) \rightarrow 1 + f(0) = f(1))$	demostrado (abrev.) desde 12
14.	$f(1 + 0) = f(1) \rightarrow 1 + f(0) = f(1)$	MP entre 8 y 13
15.	$1 + f(0) = f(1)$	MP entre 11 y 14
16.	$1 + 1 = 2$	En 15, f(0) se puede abreviar con 1 y f(1) con 2

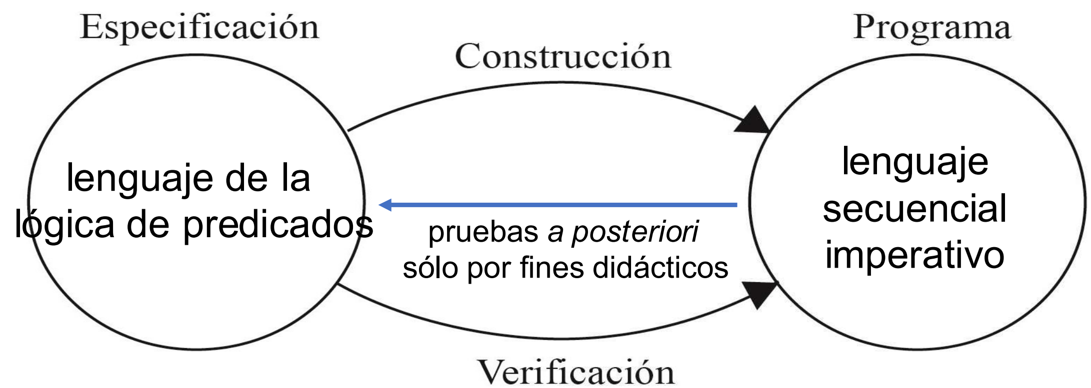
*Esta axiomática **es sensata (sound),** no produce enunciados falsos (p.ej., no permite probar 1 + 1 = 3). Pero **no es completa**, existen enunciados verdaderos que no puede probar (Teorema de Incompletitud de Gödel). Al ser incompleta también **es indecidible**: no puede decidirse la verdad o falsedad de algunos enunciados.*

Idea fuerza sostenida

Basarse en el método axiomático para contribuir a la obtención de programas correctos por construcción (esencia del desarrollo sistemático de software).



Idea fuerza: construir y verificar al mismo tiempo



Desde el punto de vista didáctico, que seguiremos, se acostumbra a presentar el método con pruebas *"a posteriori"*: dados un programa y una especificación, probar que el programa satisface la especificación.

- **Especificaciones:** formadas por predicados como true, $x + 1 = y$, $\neg(x < z)$, $x = 0 \vee x > 0$, $\exists x \forall y: x < y$, etc.
- **Programas (notación Backus-Naur)**: S :: skip | x := e | S_1 ; S_2 | if B then S_1 else S_2 fi | while B do S_1 od, siendo *e* una expresión entera y B una expresión booleana.

Primeras definiciones mediante un ejemplo

Programa S_{fac} que devuelve x! en la variable *y*

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

Especificación del programa S_{fac}

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

Se usan dos predicados, conocidos como *precondición* y *postcondición*, describiendo las condiciones de entrada y salida de S_{fac}

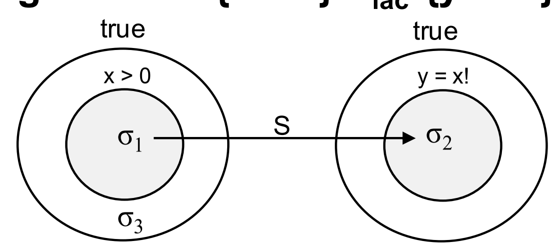
Notar que si $x < 0$, S_{fac} obtiene $y = x! = 1$, lo que es falso. ¿Esto quiere decir que el programa es incorrecto?

Terna de Hoare o fórmula de correctitud asociada

$\{x > 0\}\ S_{fac}\ \{y = x!\}$

Establece que a partir de $x > 0$, el programa S_{fac} devuelve en la variable *y* el valor x! Genéricamente se usa {p} S {q}.

Visión gráfica de $\{x > 0\}\ S_{fac}\ \{y = x!\}$



El predicado *true* denota todos los estados. A partir de todo estado $\sigma_1 \vDash x > 0$, S termina en un estado $\sigma_2 \vDash y = x!$. Y a partir de un estado $\sigma_3 \nvDash x > 0$, no importa cómo actúa S.

Estado de un programa

El estado σ corriente de un programa tiene los contenidos de todas sus variables en un momento dado. P.ej., $\sigma(a) = 1$. La expresión $\sigma \vDash p$ indica que se cumple el predicado p según σ (o σ satisface p). P.ej., si $\sigma(a)=1$, $\sigma(y)=1$, $p = (a = y)$, vale $\sigma \vDash p$.

Método axiomático

- *Contiene axiomas y reglas para cada instrucción del lenguaje de programación.*
- *Por medio de los axiomas y reglas el método permite probar fórmulas {p} S {q}.*

Método axiomático de verificación de programas

1. Axioma del skip (SKIP)	$\{p\}\ \text{skip}\ \{p\}$	El skip no modifica el estado inicial.
2. Axioma de la asignación (ASI)	$\{p[x \mid e])\}\ x := e\ \{p(x)\}$	Si vale p(x) al final, es que valía p(e) antes. ASI impone pruebas de atrás para adelante.
3. Regla de la secuencia (SEC)	$\dfrac{\{p\}\ S_1\ \{r\}\ ,\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$	El predicado r hace de nexo entre S_1 y S_2, y luego se elimina. La regla se puede generalizar a más de dos premisas.
4. Regla del condicional (COND)	$\dfrac{\{p \wedge B\}\ S_1\ \{q\}\ ,\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi}\ \{q\}}$	Único punto de entrada en que vale p, y único punto de salida en que vale q, independientemente de si B se evalúa verdadera o falsa.
5. Regla de la repetición (REP)	$\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \text{while } B \text{ do } S \text{ od}\ \{p \wedge \neg B\}}$	El predicado p es un invariante del *while*, vale antes, durante y después. A la salida del *while*, se cumple naturalmente $p \wedge \neg B$.

Notar que en un sentido los axiomas y reglas ***definen la semántica*** *de las cinco instrucciones del lenguaje.*
También notar que REP ***no asegura la terminación*** *del while (enseguida veremos cómo completar la regla).*

Ejemplo 1. Verificación de un programa de *swap* (intercambio de valores) entre dos variables.

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

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

- X e Y se conocen como *variables lógicas*, no son del programa. Por la forma de S_{swap}, recurrimos al axioma ASI tres veces, una por cada asignación, y al final completamos la prueba utilizando 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) → $\{p[x|e])\}\ x := e\ \{p(x)\}$
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) → $\dfrac{\{p\}\ S_1\ \{r\}\ ,\ \{r\}\ S_2\ \{s\}\ ,\ \{s\}\ S_3\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ ;\ S_3\ \{q\}}$

- Notar cómo el axioma ASI establece una forma de prueba **de la postcondición a la precondición**. Hay otra forma de axioma ASI, de la precondición a la postcondicíón, más complicada y menos difundida.

- También notar que obviamente se cumple la fórmula $\{y = Y \wedge x = X\}$ z := x; x := y; y := z $\{y = X \wedge x = Y\}$, pero las reglas planteadas hasta el momento no alcanzan para probarla.

  En verdad el método cuenta con una sexta regla, **la regla de consecuencia (CONS),** que permite reemplazar pre y postcondiciones por otras que las impliquen o sean implicadas por ellas, según el caso.

  En el ejemplo, como $(y = Y \wedge x = X) \longrightarrow (x = X \wedge y = Y)$, aplicando la regla CONS la prueba se completa así:

5. **$\{y = Y \wedge x = X\}$ z := x; x := y; y := z $\{y = X \wedge x = Y\}$** (4,CONS) → $\dfrac{\{p\} \longrightarrow p_1\ ,\ \{p_1\}\ S\ \{q\}}{\{p\}\ S\ \{q\}}$

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

- El siguiente programa devuelve en *y* el valor absoluto de x:

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

- Se quiere probar: **$\{true\}\ S_{va}\ \{y \geq 0\}$**

- La prueba es la siguiente:

1. $\{x \geq 0\}\ y := x\ \{y \geq 0\}$	(ASI)
2. $\{-x \geq 0\}\ y := -x\ \{y \geq 0\}$	(ASI)
3. $\{true \wedge x > 0\}\ y := x\ \{y \geq 0\}$	(1,CONS)
4. $\{true \wedge \neg(x > 0)\}\ y := -x\ \{y \geq 0\}$	(2,CONS)
5. **$\{true\}$ if $x > 0$ then $y := x$ else $y := -x$ fi $\{y \geq 0\}$**	(3,4,COND) $\longrightarrow \dfrac{\{p \wedge B\}\ S_1\ \{q\}\ ,\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{q\}}$

- En el paso 3 se reemplaza el predicado $x \geq 0$ por el predicado $true \wedge x > 0$, que lo implica.

  En el paso 4 se reemplaza el predicado $-x \geq 0$ por el predicado $true \wedge \neg(x > 0)$, que lo implica.

  Las fórmulas obtenidas en dichos pasos permiten aplicar al final la regla COND.

Ejemplo 3. Verificación de un programa que calcula el factorial.

- Ya presentamos antes el programa S_{fac} para calcular el factorial de un número natural. Se quiere probar:

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

- Se propone como invariante del *while*: $p = (y = a! \wedge a \leq x)$

- Desarrollo de la prueba:

  Prueba del fragmento previo al *while*
  1. $\{1 = a! \wedge a \leq x\}$ y := 1 $\{y = a! \wedge a \leq x\}$ (ASI)
  2. $\{1 = 1! \wedge 1 \leq x\}$ a := 1 $\{1 = a! \wedge a \leq x\}$ (ASI)
  3. $\{x > 0\}$ a := 1; y := 1 $\{y = a! \wedge a \leq x\}$ (1,2,SEC,CONS)

  Prueba del *while*
  4. $\{y \,.\, a = a! \wedge a \leq x\}$ y := y . a $\{y = a! \wedge a \leq x\}$ (ASI)
  5. $\{y \,.\, (a + 1) = (a + 1)! \wedge (a + 1) \leq x\}$ a := a + 1 $\{y \,.\, a = a! \wedge a \leq x\}$ (ASI)
  6. $\{y = a! \wedge a \leq x \wedge a < x\}$ a := a + 1; y := y . a $\{y = a! \wedge a \leq x\}$ (4,5,SEC,CONS)
  7. $\{y = a! \wedge a \leq x\}$ while a < x do a := a + 1; y := y . a od $\{y = a! \wedge a \leq x \wedge \neg(a < x)\}$ (6,REP) → $\dfrac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \text{while } B \text{ do } S \text{ od } \{p \wedge \neg B\}}$
  8. $\{y = a! \wedge a \leq x\}$ while a < x do a := a + 1; y := y . a od $\{y = x!\}$ (7,CONS)

  Paso final componiendo las dos pruebas anteriores
  9. **$\{x > 0\}$ a := 1; y := 1; while a < x do a := a + 1; y := y . a od $\{y = x!\}$** (3,8,SEC)

- Esta prueba establece que **si S_{fac} termina a partir de x > 0,** obtiene x! en la variable *y*. **Falta probar su terminación**.

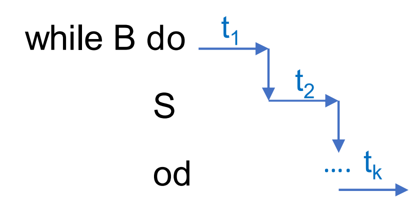
Ampliación del método axiomático de verificación de programas para probar terminación

- Para la prueba de terminación de un *while* se amplía la regla REP (para distinguirla la vamos a llamar REP*, y vamos a usar los símbolos ⟨ ⟩ en lugar de { }):

Regla de la terminación (REP*)

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

- Se le agrega un **variante**, que es una función entera t definida, como el invariante, en términos de las variables del programa.
- La variable Z es una variable lógica, no aparece en p ni en t, su objetivo es conservar el valor de t antes de la ejecución del cuerpo del *while*.
- La primera premisa es la de REP (p es el invariante).
- Por la segunda premisa, **t se decrementa en cada iteración**.
- Por la tercera premisa, **t arranca y se mantiene positiva**.
- De esta manera, el *while* debe terminar, **es imposible que haya una cadena infinita de números naturales que cumpla: $n_1 > n_2 > n_3 > \ldots$**



El valor de t decrece de iteración en iteración. Como t es una función entera positiva, indefectiblemente la cadena descendente de los t_i es finita, termina.

Ejemplo 4. Prueba de terminación del programa que calcula el factorial.

- Se quiere probar que efectivamente el programa S_{fac} **termina a partir de la precondición x > 0**:

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

- Se propone como invariante, $p = (a \leq x)$**,** más simple que el de la prueba anterior, $p = (y = a! \wedge a \leq x)$. Es que ahora la postcondición es simplemente *true,* sólo se busca probar la terminación del programa.

- Se propone como variante: $t = x - a$.
  Notar que t: al comienzo es positiva ($x > 0$ y $a = 1$), se decrementa en cada iteración (se hace $a := a + 1$ en cada iteración), y nunca se hace negativa (del *while* se sale con $x = a$).

- La prueba es la siguiente:

Para las inicializaciones (al final se cumple el invariante propuesto por primera vez):

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

Para la repetición hay que probar (a) $\langle p \wedge B\rangle\ S\ \langle p\rangle$, (b) $\langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle$, (c) $p \rightarrow t \geq 0$
siendo: $p = (a \leq x)$, $t = x - a$, $B = a < x$, $S :: a := a + 1 ;\ y := y \,.\, a$

2. $\langle a \leq x \wedge a < x\rangle\ a := a + 1 ;\ y := y \,.\, a\ \langle a \leq x\rangle$ (ASI,SEC,CONS)
3. $\langle a \leq x \wedge a < x \wedge x - a = Z\rangle\ a := a + 1 ;\ y := y \,.\, a\ \langle x - a < Z\rangle$ (ASI,SEC,CONS)
4. $a \leq x \rightarrow x - a \geq 0$ (MAT)
5. $\langle a \leq x\rangle$ while $a < x$ do $a := a + 1 ;\ y := y \,.\, a$ od $\langle a \leq x \wedge \neg(a < x)\rangle$ (2,3,4,REP*) $\longrightarrow$ $\dfrac{\langle p \wedge B\rangle\ S\ \langle p\rangle,\ \langle p \wedge B \wedge t = Z\rangle\ S\ \langle t < Z\rangle,\ p \rightarrow t \geq 0}{\langle p\rangle \text{ while } B \text{ do } S \text{ od } \langle p \wedge \neg B\rangle}$

Finalmente, considerando las 2 pruebas de arriba, se llega a:

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

Algo más sobre las especificaciones

- Una especificación establece la **relación entre los estados iniciales y finales de un programa**.

- Por ejemplo, (x = X, x = 2.X) es satisfecha por cualquier programa que duplica su entrada x. La variable x es una **variable de programa**, y la variable X es una **variable lógica** (no es parte del programa, se utiliza para fijar valores, y así para relacionar la precondición con la postcondición).

- Especificar un programa que termine con la condición x > y:
  - Una posible especificación es ($x = X \wedge y = Y$, x > y).
  - Otra más simple es (true, x > y), porque no se dice nada de x ni de *y* al principio.

- El programa del valor absoluto era: S_{va} :: if x > 0 then y := x else y := – x fi
  Antes usamos como especificación para probarlo, para simplificar: (true, $y \geq 0$).
  - No es una especificación correcta. ¿Por qué? ¿Cuál sería una especificación correcta del programa?
  - Notar por ejemplo que el programa S :: y := 0 satisface la especificación pero no es el programa esperado.
  - Una especificación correcta sería: (x = X, y = |X|).

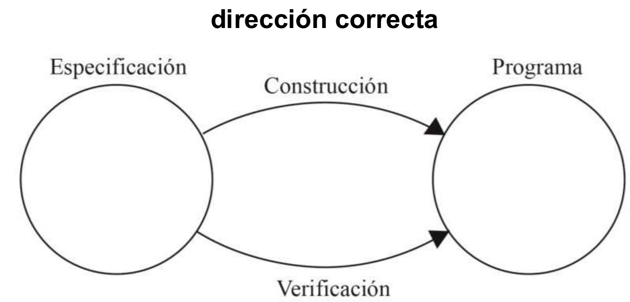
Una limitación de la lógica de predicados es que no puede expresar propiedades sobre computaciones, lo que sí puede hacer la **lógica temporal**. P.ej., la expresión G(x = 1) significa que siempre se cumple x = 1.

- Algo similar sucede con el programa del factorial: S_{fac} :: a := 1; y := 1; while a < x do a := a + 1; y := y . a od
  Antes usamos para simplificar la especificación (x > 0, y = x!).

  No es una especificación correcta. ¿Por qué? ¿Cuál sería una especificación correcta del programa?
  - El x final puede no coincidir con el inicial, pudo cambiar a lo largo de la ejecución del programa. Notar por ejemplo que el programa S :: x := 1; y := 1 satisface la especificación pero no es el programa esperado.
  - Una especificación correcta sería: ($x = X \wedge X > 0$, y = X!).

Desarrollo sistemático de programas

- **Construcción correcta** vs verificación a posteriori.



- **Idea general.**
  Supongamos que se quiere construir un programa con forma:
  $S :: S_1$; while B do S_2 od
  que satisfaga la especificación (r, q).
  Es decir, se busca: **⟨r⟩ S_1 ; while B do S_2 od ⟨q⟩.**

- En base a la metodología de pruebas definida, la construcción de S se podría encarar así:
  1. Descomponer S en sus dos componentes, **S_1** y **while B do S_2 od.**
  2. Construir **S_1** tal que **⟨r⟩ S_1 ⟨p⟩**, y **while B do S_2 od** tal que **⟨p⟩ while B do S_2 od ⟨q⟩**, siendo:
     - 2.1. p un **invariante** del while: $\langle p \wedge B\rangle\ S_2\ \langle p\rangle$.
     - 2.2. $(p \wedge \neg B) \rightarrow q$, es decir, la postcondición del while debe implicar la postcondición q de S.
     - 2.3. t un **variante** del while que decrezca en cada iteración: $\langle p \wedge B \wedge t = Z\rangle\ S_2\ \langle t < Z\rangle$.
     - 2.4. $p \rightarrow t \geq 0$, es decir, el invariante debe asegurar que el variante siempre sea positivo.

El siguiente esquema, conocido como *proof outline,* establece cómo deben **calcularse** los componentes S_1, B y S_2 del programa que se quiere construir:

⟨r⟩
S_1;
⟨p⟩
while B do
$\langle p \wedge B\rangle$
S_2
⟨p⟩
od
$\langle p \wedge \neg B\rangle$
⟨q⟩

Últimos conceptos importantes

- El método presentado es **sensato (*sound*).** Las fórmulas que prueba son verdaderas.
  - Naturalmente, no serviría un método que por ejemplo probara la fórmula $\{x = 0\}\ x := x + 1\ \{x = 2\}$.

- El método también tiene la propiedad inversa de la sensatez, es **completo**: si una fórmula es verdadera, el método permite probarla.
  - Por ejemplo, el método permite probar $\{x = 0\}\ x := x + 1\ \{x = 1\}$.

- La sensatez es una propiedad *indispensable*. La completitud, *deseable* (no siempre se cumple, depende de los lenguajes y el dominio semántico considerados por el método).

- Otra propiedad deseable es la **composicionalidad**. Se cumple en el método que estudiamos. P.ej., si se cumplen $\{p\}\ S_1\ \{q\}$ y $\{q\}\ S_2\ \{r\}$, también se cumple $\{p\}\ S_1;\ S_2\ \{r\}$, independientemente de la forma de los S_i.
  - Esta propiedad tan relevante no se cumple en la programación concurrente. Es decir, aunque se cumplan las fórmulas $\{p_1\}\ S_1\ \{q_1\}$ y $\{p_2\}\ S_2\ \{q_2\}$, no puede asegurarse que se cumpla la fórmula $\{p_1 \wedge p_2\}\ S_1 \,||\, S_2\ \{q_1 \wedge q_2\}$.

- Un programa S es **parcialmente correcto** con respecto a una especificación (p, q) sii para todo estado inicial σ que satisface p, *si S termina* lo hace en un estado σ´ que satisface q. Se anota así: **{p} S {q}.**

  Un programa S es **totalmente correcto** con respecto a una especificación (p, q) sii para todo estado inicial σ que satisface p, *S termina* y lo hace en un estado σ´ que satisface q. Se anota así: **⟨p⟩ S ⟨q⟩.**
  - Esta separación no es caprichosa, las pruebas son distintas, la correctitud parcial se prueba por inducción (ver la regla REP), mientras que la terminación no (ver la 2da y 3ra premisa de la regla REP*).