Clase teórica 13

Misceláneas de verificación de programas

Repaso de conceptos fundamentales

- Verificación **sintáctica (axiomática)** vs semántica.

- Correctitud Total = **Correctitud Parcial** + **Terminación (o No Divergencia)**. Se prueban con técnicas distintas.
  1. Correctitud parcial (método H). Terna de Hoare **{p} S {q}.** Prueba inductiva. Propiedad *safety*.
  2. Terminación (método H*). Terna de Hoare **⟨p⟩ S ⟨true⟩.** Prueba no inductiva, en base al orden bien fundado (N, <). Propiedad *liveness*.

- Una instrucción de repetición (while) concretiza dos nociones:
  1. Un **invariante** que se cumple antes, durante y después del while.
  2. Un **variante** (función que varía en (N, <)) que se decrementa luego de cada iteración del while.

- Los métodos son **composicionales**. Permiten componer ternas {p} S {q} y ⟨p⟩ S ⟨q⟩ independientemente del contenido de los S (es decir que se tratan como cajas negras).

- Los métodos son **sensatos** (propiedad imprescindible) y **completos** (propiedad deseable).

- La metodología de pruebas maneja fórmulas con **programas** y **especificaciones.** No existe la noción de programa correcto aisladamente, sino con respecto a una especificación.

- Visión de método de prueba como guía para la obtención de **programas correctos por construcción**.

Desarrollo sistemático de programas

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

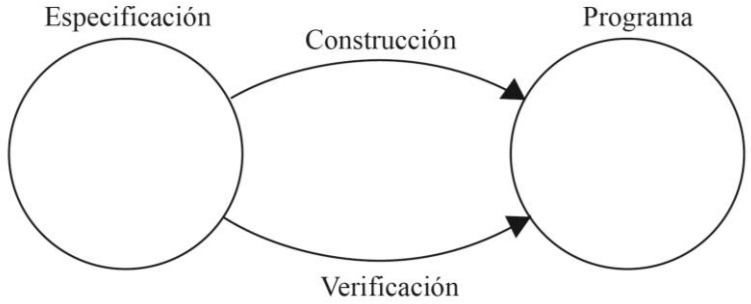
- **Idea general (Dijkstra, Gries).**

  Supongamos que se quiere construir un programa con la forma:

  T ; while B do S od

  que satisfaga la especificación:

  (r, q)



- En base a la metodología definida, la construcción del programa se podría encarar así:

1. Descomponer el programa en sus dos componentes, **T** y **while B do S od**

2. Construir T tal que **⟨r⟩ T ⟨p⟩,** y el while tal que **⟨p⟩ while B do S od ⟨q⟩,** siendo:
   - 2.1. p un **invariante** del while: $\langle p \wedge B \rangle\ S\ \langle p \rangle$.
   - 2.2. $(p \wedge \neg B) \rightarrow q$, es decir, la postcondición del while implica la postcondición q del programa.
   - 2.3. t un **variante** del while que decrece con cada iteración: $\langle p \wedge B \wedge t = Z \rangle\ S\ \langle t < Z \rangle$.
   - 2.4. $p \rightarrow t \geq 0$, es decir, el invariante asegura que el variante siempre es positivo.

⟨r⟩
T;
⟨inv p, var t⟩
while B do
⟨p ∧ B⟩
S
⟨p⟩
od
⟨p ∧ ¬B⟩
⟨q⟩

La verificación de programas y los tipos de datos

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

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

     Partiendo de i = a[2], a[1] = 2, a[2] = 2, luego de a[2] := 1 vale a[1] = 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]}

     Partiendo de a[1] = 1, x = 1, y = 1, luego de a[x] := 0 vale a[x] = a[y].

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

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

- 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**.

La verificación de programas y las estructuras de datos

- 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).

- Ejemplos (evolución):

  - **Recursos de variables compartidas**.
    Conjuntos de variables compartidas de acceso exclusivo.
    Un invariante por recurso, que vale al inicio y al final del uso del recurso.

  - **Monitor**.
    Conjuntos de variables compartidas y operaciones asociadas, de acceso exclusivo.
    Un invariante por monitor, que vale al inicio y al final del uso del monitor.

  - **Objetos**.

La verificación de programas y los procedimientos

- La regla para correctitud parcial asociada a un procedimiento **no recursivo y sin parámetros** es:

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

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

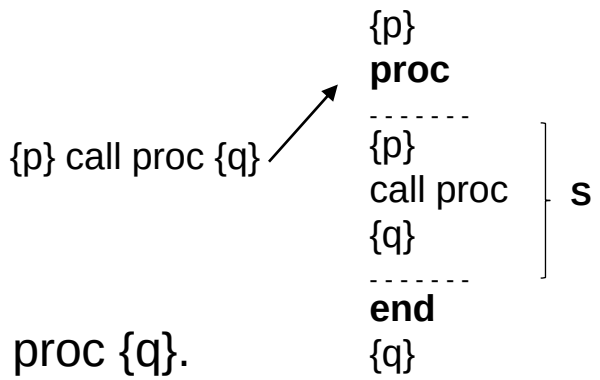
- Caso con **parámetros**. P.ej. con pasajes por valor y resultado. Se recurre a la sustitución lógica de variables:

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

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). ¿por qué q´ no incluye a la variable *y*?

- Caso de procedimiento **recursivo**:

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



Si con 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 de abajo es el que invoca a S.

Verificación de programas concurrentes

- Verificación de **más de una computación** (modelo de interleaving).
- **Más propiedades para probar**: correctitud parcial, terminación, ausencia de deadlock, exclusión mutua, ausencia de inanición.
- Distintos **modelos de comunicación**: variables compartidas, pasajes de mensajes.
- **Incompletitud**. Necesidad de **variables auxiliares**.
- Distintas hipótesis de progreso de las computaciones. **Fairness**.
- Pérdida de la **composicionalidad**. P. ej., dado $[S_1 \parallel S_2 \parallel \ldots \parallel S_n]$, una regla natural para probar la composición concurrente es la siguiente, al estilo de la regla de la secuencia para los programas secuenciales:

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

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.**

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

¿$\{q_1 \wedge q_2 \wedge q_3\}$?

Ejemplo (caso de programa con variables compartidas)

- 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.**

Un par de soluciones al problema anterior con variables compartidas

1. Cambio en la manera de especificar para obtener composicionalidad, fortaleciendo la pre y postcondición con condiciones **rely** (confiar): **qué espera un proceso del resto**, y **guarantee** (garantizar): **qué le asegura cada proceso al resto** (PROPUESTA DE JONES).

2. Prueba en **dos etapas** (PROPUESTA DE OWICKI Y GRIES).

   Etapa 1. Pruebas aisladas de cada proceso (*proof outlines*).

   Etapa 2. Chequeo global de consistencia de las pruebas de la primera etapa. Se chequea que todas las aserciones sean verdaderas, cualquiera sea el interleaving ejecutado.

   Por ejemplo, volviendo al ejemplo genérico anterior:

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

pruebas locales → prueba global

S_1 vs S_2

$\{p_1 \wedge p_2\}$ S_{21} $\{p_1\}$
$\{p_1 \wedge q_{22}\}$ S_{22} $\{p_1\}$
……
$\{p_1 \wedge q_{2k2}\}$ S_{2k2} $\{p_1\}$

y así siguiendo con:

S_1 y S_3
S_2 y S_1
S_2 y S_3
S_3 y S_1
S_3 y S_2

*Aserciones **invariantes** (proof outlines **libres de interferencia).** Posibilidad de **variables auxiliares**.*

Prueba de otras propiedades (variables compartidas, pruebas en dos etapas)

- **Terminación.** La prueba debe contemplar básicamente un aspecto adicional, no sólo la libertad de interferencia de las *proof outlines*. P.ej., si se tiene:

$$\begin{array}{llll} \langle p_1 \rangle & \langle p_2 \rangle & & \langle p_1 \rangle \\ S_1^* & S_2^* & \ldots & S^*_n \\ \langle q_1 \rangle & \langle q_2 \rangle & & \langle q_1 \rangle \end{array}$$

S_i^* es una *proof outline* del proceso S_i

también hay que chequear que el variante t del while de un proceso S_i **no sea incrementado** por una instrucción de un proceso S_j. También hay que considerar si hay alguna hipótesis de ***fairness*.** P.ej., si ninguna instrucción infinitamente habilitada para ejecutarse puede quedar postergada indefinidamente, el método de prueba debe permitir verificar la terminación de programas como el siguiente:

$$[\ \text{while } x \neq 0 \text{ do skip od} \ \|\ x := 0\ ]$$

aún si el while probado aisladamente no termina.

- **Ausencia de *deadlock*.** La prueba, como todas, parte de *proof outlines* libres de interferencia. La idea es plantear todos los casos posibles de *deadlock* y probar que las aserciones asociadas son **falsas**. P.ej., en:

$$\begin{array}{lll} & \{p_1\} & \{p_2\} \\ [\text{await } B \longrightarrow S \text{ end} & \| & T] \\ & \{q_1\} & \{q_2\} \end{array}$$

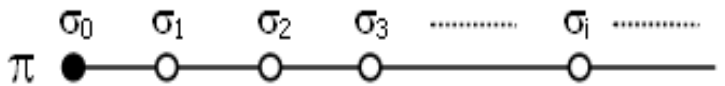
hay que probar que la aserción $p_1 \wedge \neg B \wedge q_2$ (único caso de *deadlock* en el programa) es falsa.

Anexo de la clase teórica 13

Misceláneas de verificación de programas

Verificación con lógica temporal

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



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

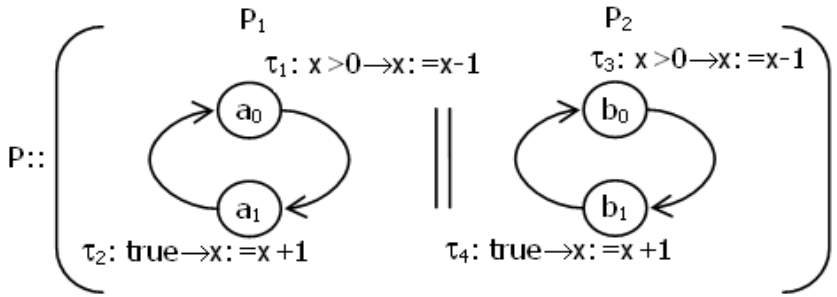
1. $\sigma_0 \models Xp$ significa que $\sigma_1 \models p$
2. $\sigma_0 \models Gp$ significa que para todo $i \geq 0$ se cumple $\sigma_i \models p$
3. $\sigma_0 \models Fp$ significa que para algún $i \geq 0$ se cumple $\sigma_i \models p$
4. $\sigma_0 \models p\ U\ q$ significa que para algún $j \geq 0$ se cumple $\sigma_j \models q$ y para todo $0 \leq i \leq j - 1$ se cumple $\sigma_i \models p$

- Hay dos familias de lenguajes de la lógica temporal, **LTL** (lógica lineal), definidos sobre computaciones (como las fórmulas presentadas más arriba), 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.

- Cuando un programa se puede modelizar con un **diagrama de transición de estados**, p.ej.:



Modelización de la solución al problema del acceso a una sección crítica por medio de un semáforo.

alcanza con la lógica temporal proposicional. Las propiedades se especifican componiendo proposiciones atómicas, y la verificación se puede llevar a cabo automáticamente **(*model checking*).** Casos típicos de programas considerados son los protocolos de comunicación y los circuitos digitales.

- **Model checking con lógica LTL.** Para verificar una fórmula F en un modelo M:

  1. Se construye un autómata (o *tableau*) A para aceptar todas las valuaciones que satisfacen ¬F.
  2. Se combina A con M, produciendo un diagrama D con caminos de A y de M.
  3. El model checker acepta sii no existe ningún camino desde el estado inicial en el diagrama combinado D (porque si existiese, habría una computación en M que no satisface la propiedad F).

- **Model checking con lógica CTL.** Para verificar F en M, se etiquetan los estados de M que satisfacen F:

  1. Se etiquetan los estados que satisfacen las subfórmulas más chicas de F.
  2. El proceso prosigue iterativamente considerando subfórmulas cada vez más grandes, hasta llegar a F.
  3. Al final se detecta si el estado inicial satisface o no F.

- Se considera que LTL es **más fácil de usar.** Como contrapartida, el model checking basado en CTL es **más eficiente**.

Soporte herramental a la verificación de programas

- 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.***
  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. Programación por ejemplos, búsqueda estocástica.

Clase práctica 13

Misceláneas de verificación de programas

Ejemplo 1. Desarrollo sistemático de un programa en base a la axiomática definida.

- Programa S_{sum} que calcule en la variable x la suma de los elementos de un arreglo de enteros a[0:N – 1] de solo lectura, con N ≥ 0.

- Por convención, si N = 0, entonces la suma es cero.

- Estructura considerada:

$$\langle r \rangle\ S_{sum}\ ::\ T\ ;\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \langle q \rangle$$

con:

$$r = N \geq 0 \quad \text{y} \quad q = (x = \Sigma_{i=0,N-1}\ a[i])$$

- Como primer paso definimos un invariante p para el while. Una estrategia conocida es **generalizar la postcondición**, reemplazando constantes por variables. En este caso reemplazamos N por una variable k. Proponemos así el siguiente invariante:

$$p = (0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i])$$

- Como segundo paso definimos B, S y un variante t para el while, para satisfacer los requerimientos planteados por la axiomática (sigue en el slide siguiente):

1. Haciendo T :: **k := 0 ; x := 0** se cumple ⟨r⟩ T ⟨p⟩.

3. Haciendo B = **k ≠ N** se cumple (p ∧ ¬B) → q.

4 y 5. Incluyendo **k := k + 1** en S y eligiendo **t = N – k,** hacemos que el variante t decrezca con cada iteración y sea ≥ 0.

2. Construimos el programa teniendo en cuenta el invariante p.

$r = N \geq 0$
$p = (0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i])$
$q = (x = \Sigma_{i=0,N-1}\ a[i])$

1. ⟨r⟩ T ; ⟨p⟩ **while B do S od** ⟨q⟩
2. ⟨p ∧ B⟩ S ⟨p⟩
3. (p ∧ ¬B) → q
4. ⟨p ∧ B ∧ t = Z⟩ S ⟨t < Z⟩
5. p → t ≥ 0

- La **proof outline** por ahora es así (sólo queda completar S):

⟨N ≥ 0⟩
k := 0 ; x := 0 ;
⟨inv: $0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i]$, var: N – k⟩
while k ≠ N do
⟨$0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i] \wedge k \neq N$⟩ *** se cumple el invariante y la guardia
S' *** descomponemos S en S´ y k := k + 1
⟨$(0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i])\ [k \mid k+1]$⟩ *** por ASI
k := k + 1
⟨$0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i]$⟩ *** se cumple el invariante
od
⟨$0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\ a[i] \wedge k = N$⟩ *** se cumple el invariante y no la guardia
⟨$x = \Sigma_{i=0,N-1}\ a[i]$⟩

- Falta completar S con S'. Debe debe satisfacer:

  $\{0 \le k \le N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k \neq N\}\ S'\ \{0 \le k + 1 \le N \wedge x = \Sigma_{i=0,k}\, a[i]\}$.

- La precondición de S' implica la aserción siguiente:

  $a_1 = (0 \le k + 1 \le N \wedge x = \Sigma_{i=0,k-1}\, a[i])$

- Y la postcondición de S' implica la aserción siguiente:

  $a_2 = (0 \le k + 1 \le N \wedge x = \Sigma_{i=0,k-1}\, a[i] + a[k])$

$\langle N \ge 0 \rangle$
k := 0 ; x := 0 ;
$\langle$inv: $0 \le k \le N \wedge x = \Sigma_{i=0,k-1}\, a[i]$, var: $N - k\rangle$
while k ≠ N do
$\langle 0 \le k \le N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k \neq N\rangle$
S'
$\langle (0 \le k \le N \wedge x = \Sigma_{i=0,k-1}\, a[i])\ [k \mid k + 1]\rangle$
k := k + 1
$\langle 0 \le k \le N \wedge x = \Sigma_{i=0,k-1}\, a[i]\rangle$
od
$\langle 0 \le k \le N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k = N\rangle$
$\langle x = \Sigma_{i=0,N-1}\, a[i]\rangle$

- Haciendo **S' :: x := x + a[k]** se cumple **$\{a_1\}$ x : = x + a[k] $\{a_2\}$**. Notar que esto no afecta el decrecimiento de t, porque t = N – k. Así concluimos la construcción de S_{sum}:

$\langle N \ge 0 \rangle$

k := 0 ; x := 0 ;

$\langle$inv: $0 \le k \le N \wedge x = \Sigma_{i=0,k-1}\, a[i]$, t: $N - k\rangle$

while k ≠ N do x := x + a[k] ; k := k + 1 od

$\langle x = \Sigma_{i=0,N-1}\, a[i]\rangle$

Ejemplo 2.

Antes probamos **|= {true} S {true}** empleando la definición de correctitud parcial, y entonces, por la completitud de H, probamos **|– {true} S {true}.**

Ahora probaremos **|– {true} S {true}** directamente, por inducción estructural, sin usar la hipótesis de completitud.

Prueba.

Base de la inducción:

S :: skip
Se cumple **|– {true} skip {true}** por el axioma SKIP.

S :: x := e
Se cumple **|– {true} x := e {true}** por el axioma ASI.

Paso inductivo:

S :: S_1 ; S_2
Por hipótesis inductiva: |– {true} S_1 {true} y |– {true} S_2 {true}.
Por SEC sobre lo anterior: **|– {true} S_1 ; S_2 {true}.**

Paso inductivo (continuación):

S :: if B then S_1 else S_2 fi

Por hipótesis inductiva: |– {true} S_1 {true} y |– {true} S_2 {true}.
Por MAT: true $\wedge$ B $\rightarrow$ true y true $\wedge \neg$B $\rightarrow$ true.
Por CONS sobre lo anterior: |– {true $\wedge$ B} S_1 {true} y |– {true $\wedge \neg$B} S_2 {true}.
Finalmente por COND sobre lo anterior: **|– {true} if B then S_1 else S_2 fi {true}.**

S :: while B do S_1 od

Por hipótesis inductiva: {true} S_1 {true}.
Por MAT: true $\wedge$ B $\rightarrow$ true.
Por CONS sobre lo anterior: {true $\wedge$ B} S_1 {true}.
Por REP sobre lo anterior: {true} while B do S_1 od {true $\wedge \neg$B}.
Por MAT: true $\wedge \neg$B $\rightarrow$ true.
Finalmente por CONS sobre lo anterior: **|– {true} while B do S_1 od {true}.**

Ejemplo 3. Redundancia de la regla AND en el método H.

- En clase se planteó el agregado de la siguiente regla AND en el método H para facilitar las pruebas:

$$\frac{\{p_1\}\ S\ \{q_1\}\ ,\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

- Por la completitud de H, la regla es **redundante**, es decir, no es necesaria, todo lo que se puede probar con ella se puede probar sinb ella utilizando sólo las reglas originales presentadas.

- Lo probaremos usando inducción estructural. Para simplificar, consideraremos un caso particular de la regla:

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

- Es decir, probaremos que para todo p, q, r, S,

si $\vdash \{p\}\ S\ \{q\}$ y $\vdash \{p\}\ S\ \{r\}$, entonces $\vdash \{p\}\ S\ \{q \wedge r\}$

sin recurrir a la regla AND.

- Base de la inducción:

S :: skip

Se tiene **|– {p} skip {q} y |– {p} skip {r}.**

Debe ser $p \rightarrow q$ y $p \rightarrow r$, y por lo tanto $p \rightarrow (q \wedge r)$.
Por SKIP: {p} skip {p}.
Finalmente por CONS sobre lo anterior: **{p} skip {$q \wedge r$}.**

S :: x := e

Se tiene **|– {p} x := e {q} y |– {p} x := e {r}.**

Debe ser $p \rightarrow q[x|e]$ y $p \rightarrow r[x|e]$, y por lo tanto $p \rightarrow (q[x|e] \wedge r[x|e])$, o lo que es lo mismo:
$p \rightarrow (q \wedge r)[x|e]$.
Por ASI: {$(q \wedge r)[x|e]$} x := e {$q \wedge r$}.
Finalmente por CONS sobre lo anterior: **{p} x := e {$q \wedge r$}.**

- Paso inductivo:

S :: S_1 ; S_2
Se tiene: **|– {p} S_1 ; S_2 {q} y |– {p} S_1 ; S_2 {r}.**
Así: (a) |– {p} S_1 {t_1}. (b) |– {t_1} S_2 {q}. (c) |– {p} S_1 {t_2}. (d) |– {t_2} S_2 {r}.
Por Hip. Ind. a partir de a y c: (e) {p} S_1 {$t_1 \wedge t_2$}.
Por CONS a partir de b y d: (f) |– {$t_1 \wedge t_2$} S_2 {q}. (g) {$t_1 \wedge t_2$} S_2 {r}.
Por Hip. Ind. a partir de f y g: (h) |– {$t_1 \wedge t_2$} S_2 {$q \wedge r$}.
Finalmente por SEC a partir de e y h: **|– {p} S_1 ; S_2 {$q \wedge r$}.**

S :: if B then S_1 else S_2 fi
Se tiene: **|– {p} if B then S_1 else S_2 fi {q} y |– {p} if B then S_1 else S_2 fi {r}.**
Así: (a) |– {$p \wedge B$} S_1 {q}. (b) |– {$p \wedge \neg B$} S_2 {q}. (c) |– {$p \wedge B$ } S_1 {r}. (d) |– {$p \wedge \neg B$ } S_2 {r}.
Por Hip. Ind. a partir de a y c, y b y d, resp.: (e) |– {$p \wedge B$} S_1 {$q \wedge r$}. (f) |– {$p \wedge \neg B$} S_2 {$q \wedge r$}.
Finalmente por COND a partir de e y f: **|– {p} if B then S_1 else S_2 fi {$q \wedge r$}.**

S :: while B do S_1 od
Se tiene: **|– {p} while B do S_1 od {q} y |– {p} while B do S_1 od {r}.**
Así: (a) |– {$p \wedge B$} S_1 {p}, con $(p \wedge \neg B) \rightarrow q$, y $(p \wedge \neg B) \rightarrow r$, por lo que: (b) $(p \wedge \neg B) \rightarrow (q \wedge r)$.
Por REP a partir de a: (c) |– {p} while B do S_1 od {$p \wedge \neg B$}.
Finalmente por CONS a partir de c y b: **|– {p} while B do S_1 od {$q \wedge r$}.**