Clase teórica 13

Verificación de programas concurrentes

Modelo 1. Lenguaje de variables compartidas

- Programas de la forma:

$$\mathbf{S :: S_0 \,;\, [S_0 \,||\, \ldots \,||\, S_n]}$$

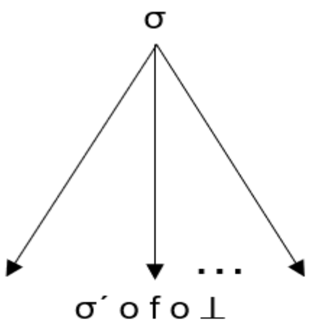
  tal que S_0 tiene inicializaciones y $[S_0 \,||\, \ldots \,||\, S_n]$ es una composición paralela de procesos $S_0, \ldots, S_n$.

- Los procesos pueden **compartir variables** y pueden **sincronizarse** con una instrucción *await*:

  await B → S end

  tal que si se cumple B, S se ejecuta atómicamente, y si no, el proceso se **bloquea** hasta que B se cumpla.

- Un programa puede tener **varias computaciones** (semántica de *interleaving*), y así, a partir de un estado inicial, puede producir **varios estados finales**:



Computaciones con 3 posibilidades

1) Finitas sin falla (estado final σ´)
2) Finitas con *deadlock* (estado de falla f)
3) Infinitas (estado indefinido ⊥)

¿En qué consiste ahora entonces la correctitud total, es decir, cuándo se cumple la fórmula **⟨p⟩ S ⟨q⟩**?

Cuando para todo estado inicial σ |= p, **toda computación** de S termina en un estado final σ´ |= q.

Ejemplos de programas del lenguaje de variables compartidas

- Programa que calcula en la variable n el **factorial** de N:

```
                    c1 := true ; c2 := true ; i := 1 ; j := N ; n := N ;
[S1 :: while c1 do await true →                    S2 :: while c2 do await true →
         if i + 1 < j then i := i + 1 ; n := n . i   ||     if j – 1 > i then j := j – 1 ; n := n . j
                    else c1 := false fi end                            else c2 := false fi end
         od                                                 od]
```

Los *awaits* se utilizan para la **exclusión mutua** entre los dos procesos.

- Programa del **productor/consumidor** (copia un arreglo *a* en un arreglo b, ambos de tamaño n, por medio de un buffer de tamaño t):

```
                    in := 0 ; out := 0 ; i := 1 ; j := 1 ;
[prod :: while i ≤ n do                            cons :: while j ≤ n do
          x := a[i] ;                                         await in – out > 0 → skip end
          await in – out < t → skip end ;     ||              y := buffer[out mod t] ;
          buffer[in mod t] := x ;                             out := out + 1 ;
          in := in + 1 ;                                      b[j] := y ;
          i := i + 1                                          j := j + 1
          od                                                  od]
```

Los *awaits* se utilizan para **sincronizar** los dos procesos (p.ej., copiar en el buffer cuando hay espacio).

Prueba de correctitud parcial

- **Ya vimos antes la pérdida de la composicionalidad en la concurrencia:**

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

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:

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

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

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

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

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

con lo que en la concurrencia dos procesos funcionalmente equivalentes **no son intercambiables.**

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

- Un par de soluciones al problema de la pérdida de la composicionalidad son:

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

2. **Prueba en dos etapas** (lo más habitual, y es lo que describiremos). **Método de Owicki y Gries**.

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

   Etapa 2. Chequeo global de consistencia de las pruebas de la primera etapa. Se chequea que todos los predicados sean verdaderos, cualquiera sea el *interleaving* ejecutado.

   Por ejemplo, considerando un programa genérico:

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

$$\xrightarrow[\text{prueba global}]{\text{pruebas locales}}$$

S_1 vs S_2

$\{p_1 \wedge p_2\}$ S_{21} $\{p_1\}$
$\{p_1 \wedge p_{22}\}$ S_{22} $\{p_1\}$
…………………..
$\{q_1 \wedge p_{2k2}\}$ S_{2k2} $\{q_1\}$

S_1 vs S_3

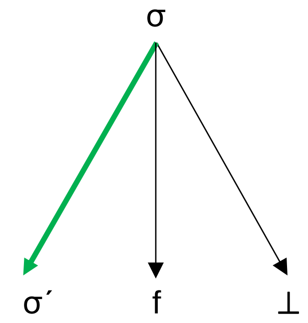
$\{p_1 \wedge p_3\}$ S_{31} $\{p_1\}$
$\{p_1 \wedge p_{32}\}$ S_{32} $\{p_1\}$
…………………..
$\{q_1 \wedge p_{3k3}\}$ S_{3k3} $\{q_1\}$

Y lo mismo con el resto de los pares

Los predicados deben ser **invariantes** (las *proof outlines deben ser* **libres de interferencia).**

- Formalizando, dada la composición $[S_1 \,||\, S_2 \,||\, \dots \,||\, S_n]$, la regla natural para la prueba de correctitud parcial es:

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



pero no es sensata. Sucede que la postcondición de una instrucción de un proceso no depende sólo de las instrucciones precedentes sino de las de los otros procesos.

- Como **aproximación al esquema composicional** se plantea (p.ej. asumiendo 3 procesos):

$$
\begin{array}{lclcl}
 & & \{p\} & & \\
\mathbf{S_1} & & \mathbf{S_2} & & \mathbf{S_3} \\
\{p_1\} & & \{p_2\} & & \{p_3\} \\
[S_{11} & & S_{21} & & S_{31} \\
\{p_{12}\} & & \{p_{22}\} & & \{p_{32}\} \\
S_{12} & & S_{22} & & S_{32} \\
\dots\dots & || & \dots\dots & || & \dots\dots \\
\{p_{1k1}\} & & \{p_{2k2}\} & & \{p_{3k3}\} \\
S_{1k1} & & S_{2k2} & & S_{3k3}] \\
\{q_1\} & & \{q_2\} & & \{q_3\} \\
 & & \{q\} & &
\end{array}
$$

con: $p \longrightarrow p_1 \wedge p_2 \wedge p_3$
$q_1 \wedge q_2 \wedge q_3 \longrightarrow q$

Regla de la composición paralela (PAR) para probar $\{p\}\ [S_1 \,||\, \dots \,||\, S_n]\ \{q\}$

- Definir *proof outlines* para cada proceso, las cuales deben cumplir:
- para todo par de *proof outlines* S_i^* y S_j^*
- para todo predicado r de S_i^*
- para toda instrucción T de S_j^* que sea una asignación o un await
- debe valer $\{r \wedge pre(T)\}\ T\ \{r\}$, siendo pre(T) la precondición de T en S_j^*

Es decir, cada predicado debe ser *invariante* cualquiera sea el *interleaving*.

Las *proof outlines* de esta forma se denominan *libres de interferencia*.

Ejemplo de prueba de correctitud parcial usando la regla PAR

- Se quiere probar: $\{x = 0\}\ [S_1 :: x := x + 1 \parallel S_2 :: x := x + 2]\ \{x = 3\}$

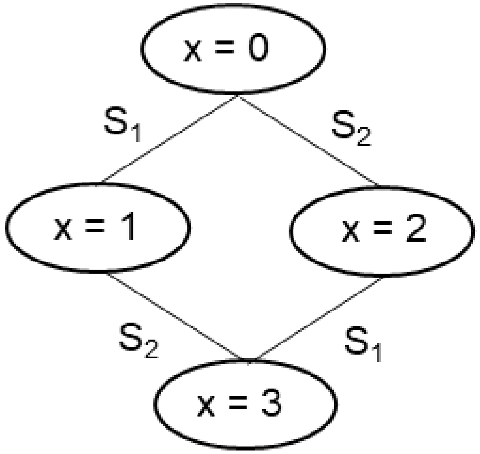
- Las *proof outlines* naturales de los procesos S_1 y S_2, considerados aisladamente, son:

$$\begin{array}{lcl} \{x = 0\} & & \{x = 0\} \\ S_1 :: x := x + 1 & \parallel & S_2 :: x := x + 2 \\ \{x = 1\} & & \{x = 2\} \end{array}$$

pero **no son libres de interferencia** (por ejemplo, al final de S_1 también puede cumplirse $x = 3$).

- Proponemos las siguientes *proof outlines*, justificadas por el diagrama que las acompaña, el cual representa los estados inicial, intermedios y final del programa y cómo se obtienen:

$$\begin{array}{ll} \{x = 0 \vee x = 2\} & \{x = 0 \vee x = 1\} \\ [S_1 :: x := x + 1 \parallel S_2 :: x := x + 2] & \\ \{x = 1 \vee x = 3\} & \{x = 2 \vee x = 3\} \end{array}$$



- Las *proof outlines* propuestas son **correctas**.

- También son **libres de interferencia**:
  las siguientes fórmulas, requeridas por la regla PAR, son verdaderas:

$$\{(x = 0 \lor x = 2) \land (x = 0 \lor x = 1)\}\ x := x + 2\ \{x = 0 \lor x = 2\}$$
$$\{(x = 1 \lor x = 3) \land (x = 0 \lor x = 1)\}\ x := x + 2\ \{x = 1 \lor x = 3\}$$
$$\{(x = 0 \lor x = 1) \land (x = 0 \lor x = 2)\}\ x := x + 1\ \{x = 0 \lor x = 1\}$$
$$\{(x = 2 \lor x = 3) \land (x = 0 \lor x = 2)\}\ x := x + 1\ \{x = 2 \lor x = 3\}$$

***proof outlines* planteadas**

$\{x = 0 \lor x = 2\}$ $\{x = 0 \lor x = 1\}$

$[S_1 :: x := x + 1 \parallel S_2 :: x := x + 2]$

$\{x = 1 \lor x = 3\}$ $\{x = 2 \lor x = 3\}$

- Y además se cumple que la precondición del programa implica la conjunción de las precondiciones de los procesos, y que la conjunción de las postcondiciones de los procesos implica la postcondición del programa:

$$\mathbf{x = 0} \longrightarrow ((x = 0 \lor x = 2) \land (x = 0 \lor x = 1))$$
$$((x = 1 \lor x = 3) \land (x = 2 \lor x = 3)) \longrightarrow \mathbf{x = 3}$$

lo que completa la prueba de **$\{x = 0\}\ [S_1 :: x := x + 1 \parallel S_2 :: x := x + 2]\ \{x = 3\}$**

Notar que para obtener *proof outlines* libres de interferencia, hay que *debilitar* los predicados utilizados en las pruebas locales. Esta técnica no siempre logra una prueba exitosa, como mostramos a continuación.

Otro ejemplo de prueba de correctitud parcial usando la regla PAR

- Se quiere probar: **$\{x = 0\}$ $[S_1 :: x := x + 1 \,||\, S_2 :: x := x + 1]$ $\{x = 2\}$**

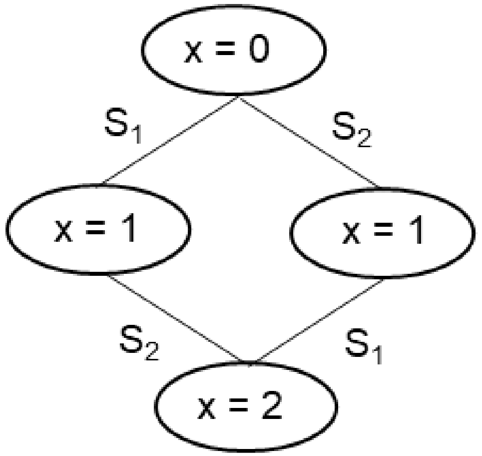
- Las *proof outlines* naturales de los procesos S_1 y S_2, considerados aisladamente, son:

$$\begin{array}{lcl} \{x = 0\} & & \{x = 0\} \\ S_1 :: x := x + 1 & || & S_2 :: x := x + 1 \\ \{x = 1\} & & \{x = 1\} \end{array}$$

pero **no son libres de interferencia** (al final de cualquiera de los procesos también se puede cumplir x = 2).

- Siguiendo el mecanismo del ejemplo anterior, proponemos las siguientes *proof outlines*:

$$\begin{array}{ll} \{x = 0 \vee x = 1\} & \{x = 0 \vee x = 1\} \\ \multicolumn{2}{l}{[S_1 :: x := x + 1 \,||\, S_2 :: x := x + 1]} \\ \{x = 1 \vee x = 2\} & \{x = 1 \vee x = 2\} \end{array}$$



- Pero tampoco dichas *proofs outlines* son libres de interferencia.

  Notar por ejemplo que la fórmula:

  $$\{(x = 0 \lor x = 1) \land (x = 0 \lor x = 1)\}\ x := x + 1\ \{x = 0 \lor x = 1\}$$

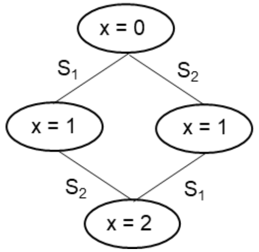
  no es verdadera.

***proof outlines* planteadas**

$\{x = 0 \lor x = 1\}$ $\quad$ $\{x = 0 \lor x = 1\}$

$[S_1 :: x := x + 1 \parallel S_2 :: x := x + 1]$

$\{x = 1 \lor x = 2\}$ $\quad$ $\{x = 1 \lor x = 2\}$



- Se puede demostrar formalmente que no existen *proof outlines* libres de interferencia en este caso.

- Intuitivamente, expresando el estado intermedio con x = 1 no alcanza para **registrar precisamente la *historia* del *interleaving* llevado a cabo**, es decir, si el que ejecutó primero $x := x + 1$ fue S_1 o S_2.

- Para resolver estos casos inevitables de **incompletitud** se contempla el uso de **variables auxiliares**. La idea es agregarlas a los procesos sin alterar el cómputo básico del programa, para **reforzar** los predicados de las *proof outlines.* En el ejemplo (se resaltan las variables auxiliares):

$$\mathbf{y := 0\ ;\ z := 0\ ;}$$

$$\{(x = 0 \land \mathbf{y = 0 \land z = 0}) \lor (x = 1 \land \mathbf{y = 0 \land z = 1})\} \qquad \{(x = 0 \land \mathbf{y = 0 \land z = 0}) \lor (x = 1 \land \mathbf{y = 1 \land z = 0})\}$$

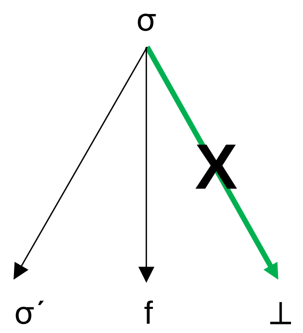
$$[S1 :: \text{await true} \rightarrow x := x + 1\ ;\ \mathbf{y := 1}\ \text{end} \parallel S2 :: \text{await true} \rightarrow x := x + 1\ ;\ \mathbf{z := 1}\ \text{end}]$$

$$\{(x = 1 \land \mathbf{y = 1 \land z = 0}) \lor (x = 2 \land \mathbf{y = 1 \land z = 1})\} \qquad \{(x = 1 \land \mathbf{y = 0 \land z = 1}) \lor (x = 2 \land \mathbf{y = 1 \land z = 1})\}$$

Se agrega *y* en S_1, z en S_2, que pasan de 0 a 1 cuando se ejecuta $x := x + 1$ (los *await* se usan para la atomicidad). Así se refuerzan los predicados y se logran *proof outlines* libres de interferencia, y como las variables auxiliares no alteran el cómputo básico, las *proof outlines prueban* el programa original.

<u>Idea general de la prueba de terminación</u> Regla PAR*

- Hay que probar que todas las computaciones sean **finitas**.



- La prueba parte también de *proof outlines* **libres de interferencia**.

  Pero debe contemplar un aspecto adicional. Dadas *proof outlines*:

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

  se debe chequear en **cada *while* de cada proceso S_i**
  que el variante t asociado **no sea incrementado** por una instrucción T de otro proceso S_j
  (sí puede ser decrementado, en cuyo caso el proceso adelanta su terminación).

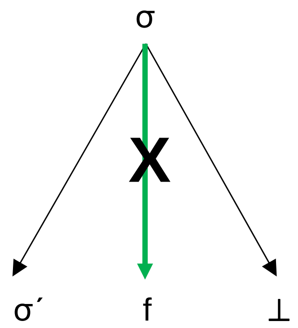
  Formalmente: **$\langle t = Z \wedge pre(T) \rangle$ T $\langle t \leq Z \rangle$, siendo pre(T) la precondición de T**

S_i	**S_j**
{t = Z}	
while	
do	{Pre(T)}
S ←	T
od	
{t ≤ Z}	

- La prueba debe contemplar también las **hipótesis de *fairness*** existentes:
  débil: "todo proceso siempre habilitado para ejecutarse se ejecuta alguna vez", o bien:
  fuerte: "todo proceso infinitamernte habilitado para ejecutarse se ejecuta alguna vez".

  P.ej., con *fairness* débil, el programa S termina,
  a pesar de que el *while* considerado aisladamente no: **S :: [while x ≠ 0 do skip od || x := 0]**

Idea general de la prueba de ausencia de *deadlock* Regla DEADLOCK



- Hay que probar que todas las computaciones terminen **sin *deadlock***.
- La prueba nuevamente parte de *proof outlines* **libres de interferencia**.
- Hay que plantear todas las posibles situaciones de *deadlock* y probar que **no ocurren.**
- Dadas *proof outlines*:

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

Paso 1. Se especifican todas las tuplas de predicados que representan potenciales casos de *deadlock*:

$$\begin{array}{c} (p_{11}, p_{12}, \ldots, p_{1n}) \\ (p_{21}, p_{22}, \ldots, p_{2n}) \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ (p_{k1}, p_{k2}, \ldots, p_{kn}) \end{array}$$

Los p_{ij} se refieren a awaits bloqueados o son postcondiciones

Paso 2. Se chequea en **cada tupla** $(p_{i1}, p_{i2}, \ldots, p_{in})$ que el predicado $p_{i1} \wedge p_{i2} \wedge \ldots \wedge p_{in}$ **sea falso**.

Por ejemplo, dado el programa:

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

la única tupla a considerar es $(p_1 \wedge \neg B, q_2)$ y el paso 2 consiste en chequear que: $(p_1 \wedge \neg B \wedge q_2)$ **sea falso**

Modelo 2. Lenguaje de recursos de variables

- También en este caso los procesos **pueden compartir variables**. Los programas son de la forma:

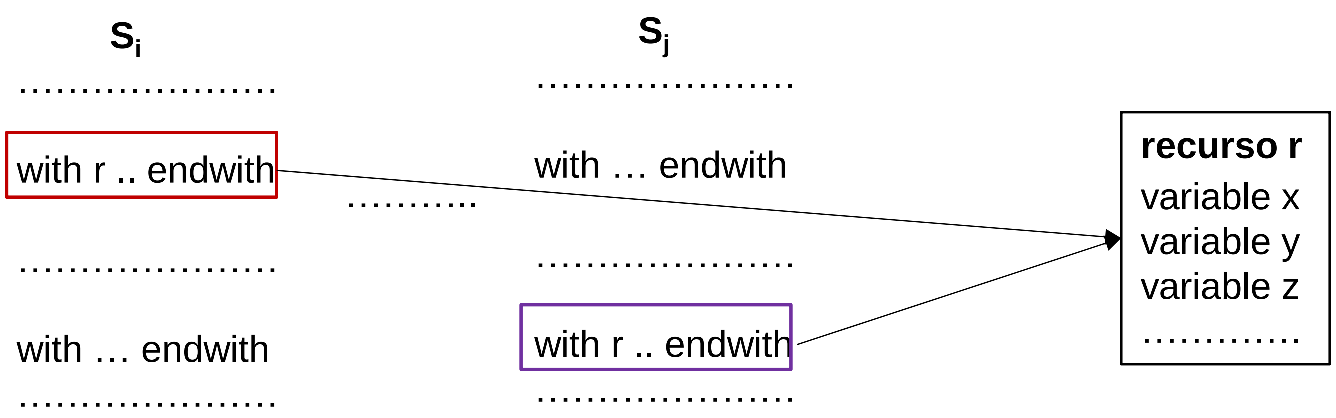
  S :: resource r_1 ($lista_1$) ; … ; resource r_m ($lista_m$) ; S_0 ; [S_1 || … || S_n]

  donde los r_i son **recursos de variables**, conjuntos disjuntos de variables definidas en las listas $lista_i$.

- Los procesos se **sincronizan** con la instrucción *with*, que provee **acceso exclusivo** a los recursos:

  with r_j when B do S endwith

  1. Cuando un proceso ejecuta una instrucción *with r_i when B do S endwith*, si r_i está libre y B es verdadera, entonces el proceso puede ocupar r_i, utilizar sus variables y progresar en su ejecución.
  2. Cuando el proceso termina de ejecutar el *with*, *libera* r_i.



Sólo un proceso accede por vez a un mismo recurso. Los *with* permiten visualizar las regiones críticas.

- La novedad en la verificación en este caso, en comparación con el modelo anterior, es el uso de **invariantes de recursos** en las *proof outlines*, los cuales se cumplen cuando los recursos **están libres.**

- P. ej., asumiendo un solo recurso r para simplificar, la **regla de correctitud parcial (RPAR)** plantea:
  - Dadas *proof outlines* $\{p_1\}\ S_1^*\ \{q_1\}, \ldots, {}_1\ \{p_n\}\ S_n^*\ \{q_n\}$ con predicados de las variables **no compartidas**,
  - y un **invariante I_r del recurso r** con las **variables compartidas**,
  - se cumple:

$$\{I_r \wedge p_1 \wedge \ldots \wedge p_n\}\ [S_1 \,\|\, \ldots \,\|\, S_n]\ \{I_r \wedge q_1 \wedge \ldots \wedge q_n\}$$

- Es decir:

$$\{I_r \wedge p_1 \wedge \ldots \wedge p_n\}$$

S_1	S_2		S_n
$\{p_1\}$	$\{p_2\}$		$\{p_n\}$
....	....		....
with … endwith	with … endwith		with … endwith
$\{p_{1i1}\}$	$\{p_{2j1}\}$		$\{p_{nk1}\}$
....	....	……	....
with … endwith	with … endwith		with … endwith
$\{p_{1i2}\}$	$\{p_{2j2}\}$		$\{p_{nk2}\}$
....	....		....

$$\{I_r \wedge q_1 \wedge \ldots \wedge q_n\}$$

De esta manera, no hay que chequear que las *proof outlines* sean libres de interferencia.

Los predicados de las proof outlines arrastran la información de las **variables disjuntas**, y el invariante del recurso arrastra la información de las **variables compartidas**.

Ejemplo de prueba de correctitud parcial de un programa que implementa un semáforo binario

$\{0 \le s \le 1\}$

[with sem when s = 1 do s := 0 endwith ; … sección 1 … with sem when true do s := 1 endwith || with sem when s = 1 do s := 0 endwith ; … sección 2 … with sem when true do s := 1 endwith]

$\{0 \le s \le 1\}$

- Para probar que el programa cumple (**s = 1, s = 1),** hay que reforzar las *proof outlines* con **variables auxiliares**:

$$\{0 \le s \le 1 \wedge ((a = 0 \wedge b = 0) \vee (a = 1 \wedge b = 0) \vee (a = 0 \wedge b = 1)) \wedge ((a = 0 \wedge b = 0) \rightarrow s = 1) \wedge (((a = 1 \wedge b = 0) \vee (a = 0 \wedge b = 1)) \rightarrow s = 0)\}$$

a := 0 ; b := 0 ;

{a = 0}
[with semáforo when s = 1 do
s := 0 ; **a := 1** endwith ; **{a = 1}**
… sección 1 … **{a = 1}**
with semáforo when true do
s := 1 ; **a := 0** endwith
{a = 0}

||

{b = 0}
with semáforo when s = 1 do
s := 0 ; **b := 1** endwith ; **{b = 1}**
… sección 2 … **{b = 1}**
with semáforo when true do
s := 1 ; **b := 0** endwith]
{b = 0}

$$\{0 \le s \le 1 \wedge ((a = 0 \wedge b = 0) \vee (a = 1 \wedge b = 0) \vee (a = 0 \wedge b = 1)) \wedge ((a = 0 \wedge b = 0) \rightarrow s = 1) \wedge (((a = 1 \wedge b = 0) \vee (a = 0 \wedge b = 1)) \rightarrow s = 0)\}$$

- I_{sem} es el invariante del semáforo. **Al final se cumple: $a = 0 \wedge b = 0 \wedge I_{sem}$, que implica s = 1.**

Las pruebas de terminación y ausencia de *deadlock* son como en el modelo anterior, pero con invariantes de recursos

Modelo 3. Lenguaje de pasajes de mensajes

- Las composiciones concurrentes tienen **procesos etiquetados**:

$$[P_1 :: S_1 \,||\, \ldots \,||\, P_n :: S_n]$$

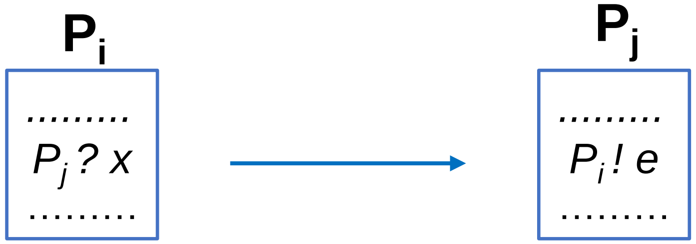
- Los procesos **no comparten variables** y se envían **mensajes** con **instrucciones de comunicación**:

- ***Instrucción de entrada P_j ? x***

  En un proceso P_i con $i \neq j$, que tiene una variable local x, la instrucción efectúa un pedido al proceso P_j para que asigne un valor a x. P_i queda **bloqueado** mientras P_j no responda.

- ***Instrucción de salida P_i ! e***

  En un proceso P_j con $j \neq i$, la instrucción efectúa un pedido al proceso P_i para que reciba el valor de la expresión *e*, definida con variables locales de P_j. P_j queda **bloqueado** mientras P_i no responda.



Los pasajes de mensajes son sincrónicos

- Las instrucciones de comunicación pueden usarse aisladamente o en **condicionales** y **repeticiones**. P.ej.:

P_i :: **if** $x > 0$; P_j ? y $\rightarrow$ $y := y + 1$ **or** $x < 0$; P_j ! 0 $\rightarrow$ skip **fi**, P_j :: **do** $x \geq 0$; P_i ! x $\rightarrow$ $x := x - 1$ **od**,   etc.

Idea general de las pruebas en este modelo. **Método de Apt, Francez y de Roever.**

- Sea el siguiente programa:

P1 || P2 || P3], con:
P1 :: P2 ! y1
P2 :: P1 ? y2 ; P3 ! y2
P3 :: P2 ? y3

El programa consiste en propagar un valor desde el proceso P_1 hasta el proceso P_3.

- Siguiendo con la metodología de dos etapas (pruebas locales y chequeo global de consistencia), la novedad en este modelo es que en las pruebas locales hay que utilizar **asunciones**:

***Axioma de la instrucción de entrada* (IN)**
{p} Pi ? x {q}

***Axioma de la instrucción de salida* (OUT)**
{p} Pj ! e {p}

- La segunda etapa consiste en un **test de cooperación** en el que se validan **todas** las asunciones de las pruebas locales. Para ello se utiliza el siguiente axioma:

***Axioma de comunicación* (COM)**
{p[x|e]} Pi ? x || Pj ! e {p}

¿A qué axioma recuerda?
Al axioma de **asignación**

- Finalmente, la **regla de prueba de correctitud parcial (DPAR)** tiene la forma habitual: dadas *proof outlines* $\{p_1\}\ S_1^*\ \{q_1\}, \ldots, \{p_n\}\ S_n^*\ \{q_n\}$ *que **cooperan*** (en términos del axioma COM), se cumple $[P_1 \,||\, \ldots \,||\, P_n]$.

- Veamos cómo probamos la correctitud parcial del programa anterior aplicando la regla DPAR:

Pruebas locales con asunciones (etapa 1)

$P_1 :: \{y_1 = Y\}$ $\mathbf{P_2 \,!\, y_1}$ $\{y_1 = Y\}$
$P_2 :: \{true\}$ $\mathbf{P_1 \,?\, y_2}$ $\{y_2 = Y\}$; $\mathbf{P_3 \,!\, y_2}$ $\{y_2 = Y\}$
$P_3 :: \{true\}$ $\mathbf{P_2 \,?\, y_3}$ $\{y_3 = Y\}$

Test de cooperación (etapa 2)

$\{y_1 = Y \wedge true\}$ $\mathbf{P_2 \,!\, y_1 \,||\, P_1 \,?\, y_2}$ $\{y_1 = Y \wedge y_2 = Y\}$
$\{y_2 = Y \wedge true\}$ $\mathbf{P_3 \,!\, y_2 \,||\, P_2 \,?\, y_3}$ $\{y_2 = Y \wedge y_3 = Y\}$

Finalmente, de acuerdo a la regla DPAR queda:

$$\{y_1 = Y \wedge true \wedge true\}\ [P_1 \,||\, P_2 \,||\, P_3]\ \{y_1 = Y \wedge y_2 = Y \wedge y_3 = Y\}$$

y por la regla de consecuencia (CONS):

$$\{y_1 = Y\}\ [P_1 \,||\, P_2 \,||\, P_3]\ \{y_3 = Y\}$$

- Sea ahora este otro programa: $[P_1 \| P_2]$, con:

  $P_1 :: x := 0 \; ; \; P_2 ! \, x \; ; \; x := x + 1 \; ; \; P_2 ! \, x$

  $P_2 :: P_1 ? \, y \; ; \; P_1 ? \, y$

  P_1 primero le envía un 0 y luego un 1 a P_2, quedando la variable y de P_2 en 1.

- Las *proof outlines* naturales son:

  $P_1 :: \{true\} \; x := 0 \; \{x = 0\} \; ; \; P_2 ! \, x \; \{x = 0\} \; ; \; x := x + 1 \; \{x = 1\} \; ; \; P_2 ! \, x \; \{x = 1\}$

  $P_2 :: \{true\} \; P_1 ? \, y \; \{y = 0\} \; ; \; P_1 ? \, y \; \{y = 1\}$

  pero mientras que se cumple:

  $\{x = 0 \wedge true\} \; P_2 ! \, x \, \| \, P_1 ? \, y \; \{x = 0 \wedge y = 0\}$

  $\{x = 1 \wedge y = 0\} \; P_2 ! \, x \, \| \, P_1 ? \, y \; \{x = 1 \wedge y = 1\}$

  no se cumple:

  $\{x = 0 \wedge y = 0\} \; P_2 ! \, x \, \| \, P_1 ? \, y \; \{x = 0 \wedge y = 1\}$ (*)

  $\{x = 1 \wedge true\} \; P_2 ! \, x \, \| \, P_1 ? \, y \; \{x = 1 \wedge y = 0\}$ (**)

  lo que es lógico porque tales comunicaciones **no ocurren**.

- Para evitar estos cruces inexistentes se usan **invariantes globales**. La idea es, siguiendo con el ejemplo:

1. Agregamos variables $\mathbf{c_1}$ en P_1 y $\mathbf{c_2}$ en P_2, de valor inicial **0**, que se incrementan en **1** cuando los procesos hacen una comunicación. Así, **c_1 y c2 son iguales antes y después de cada comunicación**.
2. Definimos como invariante global: **IG = ($c_1 = c_2$),** lo que permite probar trivialmente las fórmulas (*) y (**):
   - En la fórmula (*), $c_1 = 1$ y $c_2 = 2$, por lo que **la precondición IG ∧ x = 0 ∧ y = 0 es falsa.**
   - En la fórmula (**), c1 = 2 y c2 = 1, por lo que **la precondición IG ∧ x = 1 ∧ true es falsa.**

Acerca de la prueba de terminación

- Un proceso de la forma, p.ej., **P_1 :: do x ≠ 0 ; P_2 ? x → skip od**, termina si recibe alguna vez el valor 0.

- En este caso, no sólo hay que recurrir a una asunción, sino que además es necesario inicializar el variante de la repetición con algún valor **desconocido** (representando la máxima cantidad posible de iteraciones).
  La solución matemática en este caso es asignar el primer ordinal infinito, ω, mayor que todos los naturales.

- Sea este otro programa [P_1 || P_2], con precondición **$0 \leq x \leq 10 \land y > 0$:**

  P1 :: do x ≥ 0 ; P2 ! x → x := x – 1 od
  P2 :: do y ≠ 0 ; P1 ? y → skip od

  Ejercicio. ¿Qué valor inicial puede asignarse a los variantes de las repeticiones?

- Como en los métodos anteriores, la prueba de terminación puede contemplar **hipótesis de *fairness***. P. ej.:

```
[P1 :: b := true ;                        P2 :: c := true ;
       do b ; P2 ? b → skip od      ||           do c ; P1 ! true → skip
                                                 or c ; P1 ! false → c := false od]
```

  En la instrucción de repetición de P_2 cada vez se puede tomar una entre dos direcciones.

  El programa termina sólo si en P_2 se elige alguna vez la segunda dirección.

  Esto sólo se asegura si el *fairness* es de *nivel comunicación*: toda comunicación infinitamente habilitada se ejecuta alguna vez.

Acerca de la prueba de ausencia de deadlock

- Se mantiene la idea de plantear todos los casos posibles de *deadlock* y probar que no ocurre ninguno.
- Obviamente, toda tupla debe considerar **al menos una instrucción de comunicación**.
- P.ej., volviendo al programa **{y1 = Y}** **[P1 :: P2 ! y1 || P2 :: P1 ? y2 ; P3 ! y2 || P3 :: P2 ? y3]** **{y3 = Y},** antes probamos su correctitud parcial con las *proof outlines*:

{y1 = Y} P2 ! y1 {y1 = Y}
{true} P1 ? y2 {y2 = Y}; P3 ! y2 {y2 = Y}
{true} P2 ? y3 {y3 = Y}

- Así planteadas, no sirven para probar la ausencia de *deadlock*, hay que utilizar **variables auxiliares** y un **invariante global.** P.ej., el predicado que representa el *deadlock* “P_1 terminó y P_2 y P_3 no empezaron” es: $\mathbf{y_1 = Y \land true \land true}$, que no es falso.
- Proponemos:
  (los delimitadores <>
  denotan atomicidad)

{w1 = 1}		**{w2 = 0}**		**{w3 = 0}**
[< P2 ! y1 ; w1 := 2>	**||**	**<P1 ? y2 ; w2 := 1>**	**||**	**<P2 ? y3 ; w3 := 2>]**
{w1 = 2}		**{w2 = 1}**		**{w3 = 2}**
		<P3 ! y2 ; w2 := 2>		
		{w2 = 2}		

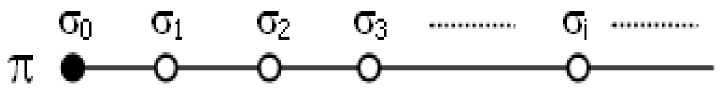
y el invariante global **IG = (w1 = 1 ∧ w2 = 0 ∧ w3 = 0) ∨ (w1 = 2 ∧ w2 = 1 ∧ w3 = 0) ∨ (w1 = 2 ∧ w2 = 2 ∧ w3 = 2).**
tal que wi = 0 significa que Pi no empezó, wi = 1 que Pi está por enviar un valor, y wi = 2 que Pi terminó.

Ejercicio. Dar todos los casos de deadlock con las variables wi y el invariante IG y mostrar que no se cumplen.

Anexo

Verificación de programas concurrentes utilizando 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:

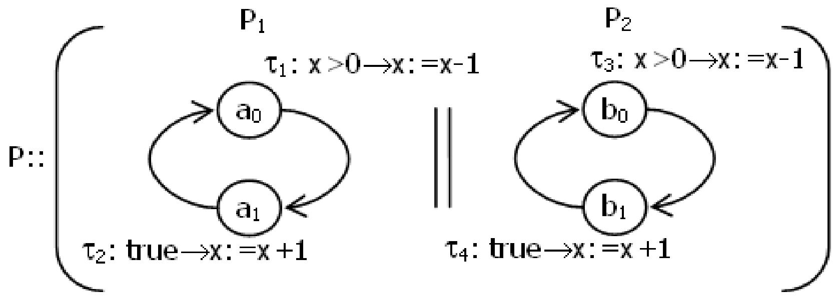
1. σ_0 |= Xp significa que σ_1 |= p
2. σ_0 |= Gp significa que para todo $i \geq 0$ se cumple σ_i |= p
3. σ_0 |= Fp significa que para algún $i \geq 0$ se cumple σ_i |= p
4. σ_0 |= p U q significa que para algún $j \geq 0$ se cumple σ_j |= q y para todo $0 \leq i \leq j - 1$ se cumple σ_i |= p

- Hay dos familias de lenguajes de 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. σ_0 |= AGp significa que sobre toda computación desde σ_0 se cumple Gp
2. σ_0 |= 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.

- 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 A para aceptar todas las valuaciones que satisfacen $\neg$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 (si existiese, habría una computación en M que no satisface la propiedad F).

- **Model checking con lógica CTL.** Para verificar una fórmula 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**.

Metodología UNITY

- **UNITY** es una metodología de especificación, desarrollo y verificación de programas concurrentes, utilizando una variante de la lógica temporal (**Método de Chandy y Mizra**).

- Dos estrategias de desarrollo, **unión** (composición) y **superposición** (refinamiento).

- La unión de dos procesos S1 y S2 consiste en su **concatenación** [S1 || S2], y la superposición establece una **programación por capas** (se agregan variables y asignaciones que no alteran los cómputos inferiores).

- Las especificaciones cuentan con **operadores temporales** como:

  1. **p *unless* q**: si se cumple p entonces q no se cumple nunca y p se cumple siempre, o q se cumple a futuro y p se cumple mientras q no se cumpla.
  2. **p *ensures* q**: si se cumple p entonces q se cumple a futuro y p se cumple mientras q no se cumpla.
  3. **p *leads-to* q**: si se cumple p entonces q se cumple a futuro.

- Se demuestra, dada la unión [S1 || S2], que:

  a. Si se cumple **p *unless* q en S1 y S2**, se cumple **p *unless* q en [S1 || S2].**
  b. Si se cumple **p *ensures* q en S1** y **p *unless* q en S2**, se cumple **p *ensures* q en [S1 || S2].**
  c. Si se cumple ***p leads-to* q en S1 y S2**, no tiene por qué cumplirse ***p leads-to* q en [S1 || S2].**

- Por su parte, la superposición facilita la verificación, pero como contrapartida, exige conocer en detalle las capas previamente desarrolladas, al tiempo que su tratamiento algebraico es limitado.

Clase práctica 13

Ejemplo 1. Desarrollo sistemático de un programa con la guía de la Lógica de Hoare.

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

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

- Especificación y estructura considerada:

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

  tal que:

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

- 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):

En base a la estrategia basada en la Lógica de Hoare:

1. Hacemos T :: **k := 0 ; x := 0.** Se cumple ⟨r⟩ T ⟨p⟩.

3. Hacemos B = **k ≠ N.** Se cumple $(p \wedge \neg B) \rightarrow q$.

4 y 5. Incluimos **k := k + 1** en S y elegimos **t = N – k.**
Se cumple que el variante t decrece en cada iteración y es ≥ 0.

2. Falta completar S para que desde $p \wedge B$ se mantenga p (se ve después).

- La **proof outline** por ahora queda así:

$\langle N \geq 0 \rangle$
k := 0 ; x := 0 ;
$\langle \text{inv: } 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i]\ ,\ \text{var: } t = N - k \rangle$
while k ≠ N do
$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k \neq N \rangle$
S' → descomponemos S en S´ y k := k + 1
$\langle \mathbf{0 \leq k + 1 \leq N \wedge x = \Sigma_{i=0,k}\, a[i]} \rangle$ → por el axioma de asignación
k := k + 1
$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \rangle$
od
$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k = N \rangle$
$\langle x = \Sigma_{i=0,N-1}\, a[i] \rangle$

Estrategia

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

Construir un programa tal que:
1. ⟨r⟩ T ; ⟨p⟩ while B do S od ⟨q⟩
2. $\langle p \wedge B \rangle$ S ⟨p⟩
3. $(p \wedge \neg B) \rightarrow q$
4. $\langle p \wedge B \wedge t = Z \rangle$ S $\langle t < Z \rangle$
5. $p \rightarrow t \geq 0$

- Completamos ahora S. Se debe cumplir:

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

a) La precondición de S' implica la aserción siguiente:

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

b) La postcondición de S' implica la aserción siguiente:

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

c) Haciendo **S' :: x := x + a[k]** se cumple **$\{a_1\}$ x : = x + a[k] $\{a_2\}$**

y **S' no afecta el decrecimiento de t,** porque t = N – k. Así concluimos la construcción de S_{sum}:

$\langle N \geq 0\rangle$

k := 0 ; x := 0 ;

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

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

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

Proof outline hasta el momento

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

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: $\vdash$ {true} S_1 {true} y $\vdash$ {true} S_2 {true}.
Por MAT: true $\wedge$ B $\rightarrow$ true y true $\wedge$ $\neg$B $\rightarrow$ true.
Por CONS sobre lo anterior: $\vdash$ {true $\wedge$ B} S_1 {true} y $\vdash$ {true $\wedge$ $\neg$B} S_2 {true}.
Finalmente por COND sobre lo anterior: **$\vdash$ {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: **$\vdash$ {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 sin 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 **$\vdash$ {p} skip {q} y $\vdash$ {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 **$\vdash$ {p} x := e {q} y $\vdash$ {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: **$\vdash$ {p} S_1 ; S_2 {q} y $\vdash$ {p} S_1 ; S_2 {r}.**

Así: (a) $\vdash \{p\}\ S_1\ \{t_1\}$, (b) $\vdash \{t_1\}\ S_2\ \{q\}$, (c) $\vdash \{p\}\ S_1\ \{t_2\}$, (d) $\vdash \{t_2\}\ S_2\ \{r\}$.

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) $\vdash \{t_1 \wedge t_2\}\ S_2\ \{q\}$, (g) $\{t_1 \wedge t_2\}\ S_2\ \{r\}$.

Por Hip. Ind. a partir de f y g: (h) $\vdash \{t_1 \wedge t_2\}\ S_2\ \{q \wedge r\}$.

Finalmente por SEC a partir de e y h: **$\vdash \{p\}\ S_1 ; S_2\ \{q \wedge r\}$.**

S :: if B then S_1 else S_2 fi

Se tiene: **$\vdash$ {p} if B then S_1 else S_2 fi {q} y $\vdash$ {p} if B then S_1 else S_2 fi {r}.**

Así: (a) $\vdash \{p \wedge B\}\ S_1\ \{q\}$, (b) $\vdash \{p \wedge \neg B\}\ S_2\ \{q\}$.
(c) $\vdash \{p \wedge B\}\ S_1\ \{r\}$, (d) $\vdash \{p \wedge \neg B\}\ S_2\ \{r\}$.

Por Hip. Ind. a partir de a y c, y b y d: (e) $\vdash \{p \wedge B\}\ S_1\ \{q \wedge r\}$, (f) $\vdash \{p \wedge \neg B\}\ S_2\ \{q \wedge r\}$.

Finalmente por COND a partir de e y f: **$\vdash$ {p} if B then S_1 else S_2 fi $\{q \wedge r\}$.**

S :: while B do S_1 od

Se tiene: **$\vdash$ {p} while B do S_1 od {q} y $\vdash$ {p} while B do S_1 od {r}.**

Así: (a) $\vdash \{p \wedge B\}\ S_1\ \{p\}$, con $(p \wedge \neg B) \rightarrow q$ y $(p \wedge \neg B) \rightarrow r$, y entonces:
(b) $(p \wedge \neg B) \rightarrow (q \wedge r)$.

Por REP a partir de a: (c) $\vdash$ {p} while B do S_1 od $\{p \wedge \neg B\}$.

Finalmente por CONS a partir de b y c: **$\vdash$ {p} while B do S_1 od $\{q \wedge r\}$.**