

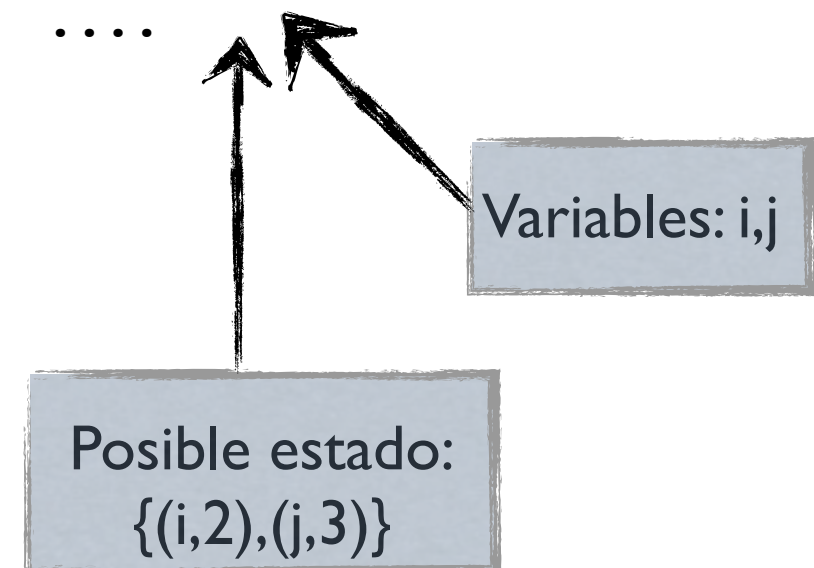
Programación Imperativa

Programación Avanzada-UNRC-2011
Pablo Castro

Estados y Predicados

- La programación imperativa se basa en la noción de estado y variables.
- Cada programa imperativo utiliza un conjunto de variables: x, y, z, \dots
- Un estado en la ejecución de un programa imperativo es una asignación de valores a sus variables.
- Las instrucciones nos permiten cambiar de estados.

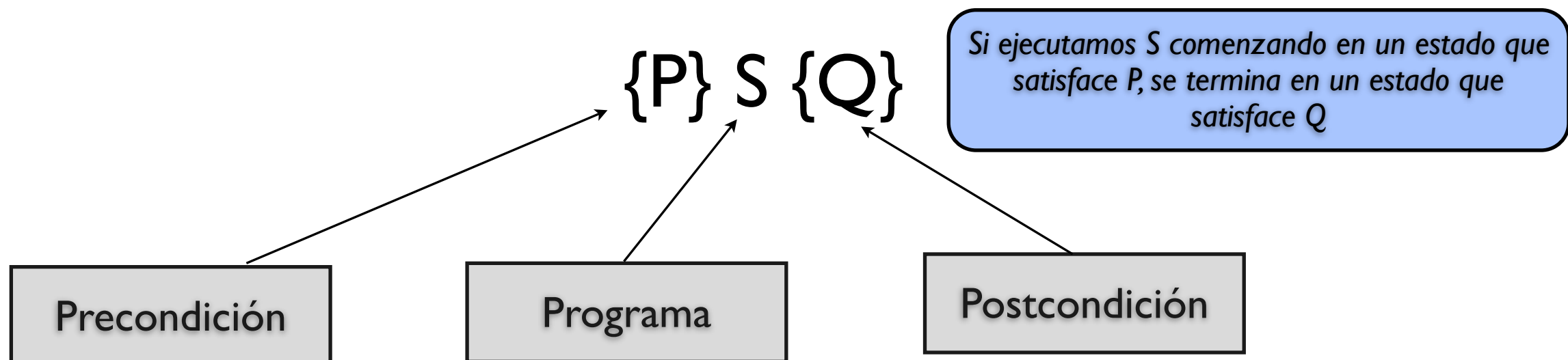
```
program example(output);  
var i : integer;  
var j : integer;  
....
```



Especificando Programas Imperativos

- A Diferencia de funcional la ejecución de un programa imperativo produce cambios de estado.
- En este caso una especificación indica el estado en que esperamos que termine un programa (**postcondición**), dada cierta condición inicial (**precondición**).

Escribimos una especificación de la siguiente forma:



Pre y Postcondiciones

- Las pre/post-condiciones son predicados lógicos (proposicionales o de primer orden).
- Los programas son escritos en cualquier lenguaje imperativo: Pascal, C, Java, etc.

Ejemplo:

$\{\text{True}\}S\{x=y\}$

El programa S debe terminar satisfaciendo $x=y$, para cualquier valor de x e y

Estados y Predicados

Dado un estado s , diremos que:

$$s \models P$$

Cuando los valores asignados por s a las variables hacen P verdadero.

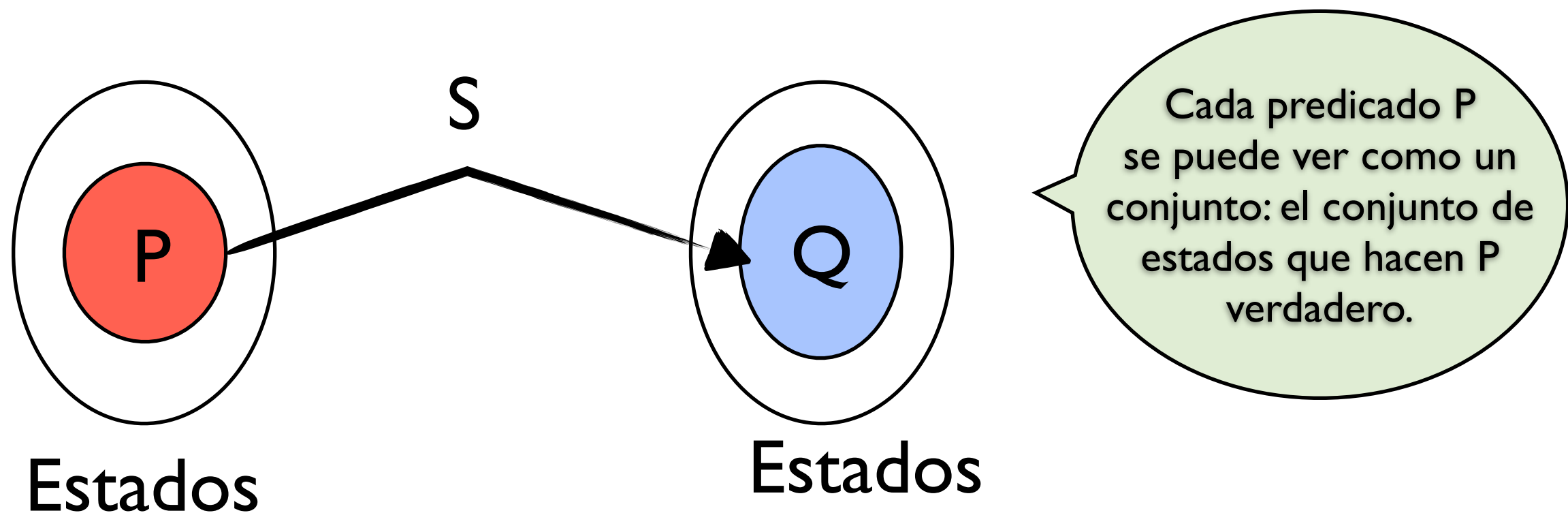
$$\{(x, 5), (y, 10)\} \models y = x + 5$$

Estado

Predicado

Ternas de Hoare y Conjuntos

Podemos interpretar $\{P\}S\{Q\}$ utilizando conjuntos:



$\{P\}S\{Q\}$ se cumple si siempre que partimos de algún estado del conjunto P, se llega por S a un estado que pertenece a Q

Predicados Universalmente Validos

Diremos que:

[P]

P es un predicado

Cuando:

$\langle \forall s : s \text{ es estado} : s \models P \rangle$

Todo estado satisface P

Por ejemplo:

$[0 = 1 \Rightarrow x = y]$

Es verdadero
independientemente de los valores
de x e y


Ternas de Hoare

Una terna de Hoare es:

$$\{P\}S\{Q\}$$

Estas ternas cumplen ciertas propiedades lógicas:

- Exclusión de milagros: $\{P\}S\{False\} \equiv [P \equiv False]$



No hay ningún estado del cual podemos empezar, y terminar satisfaciendo False.

Propiedades de la ternas de Hoare

- Fortalecimiento de la precondición:

$$\{P\}S\{Q\} \wedge [P_0 \Rightarrow P] \Rightarrow \{P_0\}S\{Q\}$$

$$\{x = 2 \vee x = 5\}S\{Q\} \Rightarrow \{x = 2\}S\{Q\}$$

Si el programa S, cuando empieza en un estado satisfaciendo P, termina en un estado satisfaciendo Q, y P_0 es más fuerte que P. Entonces si empezamos en un estado satisfaciendo P_0 , también garantizaremos Q

- Debilitamiento de la postcondición:

$$\{P\}S\{Q\} \wedge [Q \Rightarrow Q_0] \Rightarrow \{P\}S\{Q_0\}$$

$$\{P\}S\{x = 2\} \Rightarrow \{P\}S\{x = 2 \vee x = 5\}$$

Si el programa S, cuando empieza en un estado satisfaciendo P, termina en un estado satisfaciendo Q, y Q_0 es más débil que Q. Entonces si empezamos en un estado satisfaciendo P, también garantizaremos Q_0 .

Propiedades de la ternas de Hoare (cont.)

- Conjunción de postcondición:

$$\{P\}S\{Q\} \wedge \{P\}S\{Q'\} \equiv \{P\}S\{Q \wedge Q'\}$$

Si el programa S, cuando empieza en un estado satisfaciendo P, podemos asegurar postcondiciones Q y Q'. Entonces podemos asegurar $Q \wedge Q'$

- Disyunción de la precondición:

$$\{P\}S\{Q\} \wedge \{P'\}S\{Q\} \equiv \{P \vee P'\}S\{Q\}$$

Dado un estado satisfaciendo P, S garantiza Q, y dado un estado satisfaciendo P', S también garantiza Q. Entonces, partiendo de un estado satisfaciendo $P \vee P'$, S también garantiza Q.

El Transformador de predicados wp

Para cada comando S , podemos definir una función

$$wp.S : Predicados \rightarrow Predicados$$

Dado un predicado Q , $wp.S$ devuelve el predicado (precondición) más débil que cumple:

$$\{WP.S.Q\} S \{Q\}$$

Es decir, $wp.S.Q$ cumple:

1. $\{wp.S.Q\} S \{Q\}$
2. $\langle \forall P' :: \{P'\} S \{Q\} \Rightarrow (P' \Rightarrow wp.S.Q) \rangle$

El Transformador de predicados WP (cont)

La función wp nos permite probar $\{P\}S\{Q\}$:

$$\{P\}S\{Q\} \equiv [P \Rightarrow wp.S.Q]$$

Usando esta propiedad podemos decir que un programa $\{P\}S\{Q\}$ es correcto cuando se cumple: $[P \Rightarrow wp.S.Q]$

Podemos escribir las reglas anteriores de la siguiente forma:

1. $[wp.S.False \equiv False]$
2. $[wp.S.Q \wedge wp.S.R \equiv wp.S.(Q \wedge R)]$
3. $[wp.S.Q \vee wp.S.R \Rightarrow wp.S.(Q \vee R)]$

Un pequeño lenguaje Imperativo

Tendremos 3 clases de variables:

- *Constantes*: sirven para hablar de los valores de las variables.
- *Variables* de programación.
- *Variables* cuantificadas.

```
begin
  cons X,Y: Int;
  var x,y: Int;
  {  $X > 0 \wedge Y > 0 \wedge x = X \wedge y = Y$  }
  S
  {  $x = \text{mcd}.X.Y$  }
end
```

La sentencia *skip*

La sentencia *skip* nos devuelve el mismo estado:

- $[wp.skip.Q \equiv Q]$
- $\{P\}skip\{Q\} \equiv [P \Rightarrow Q]$ ←

El programa:

```
begin
  var x: Int;
  {x ≥ 1}
  skip
  {x ≥ 0}
end
```

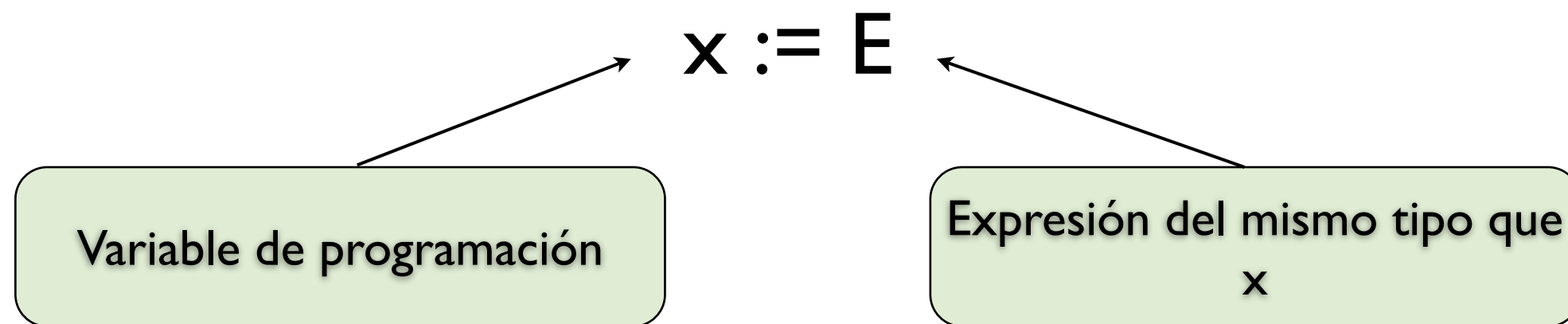
es correcto ya que:

$$[x \geq 1 \Rightarrow x \geq 0]$$

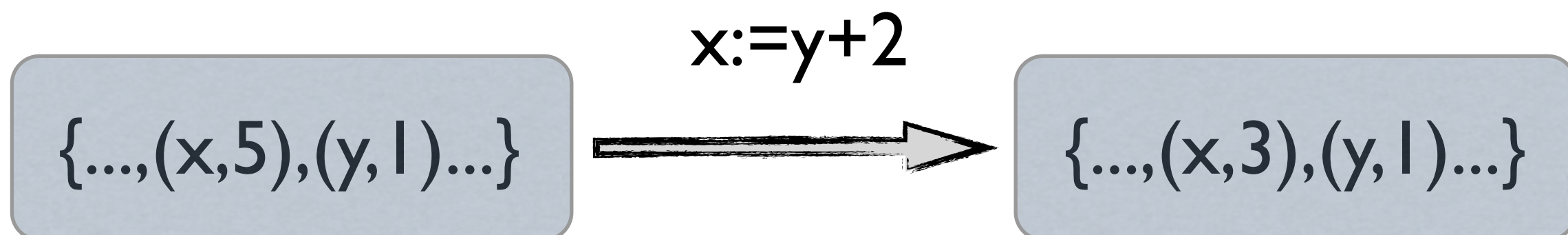
El *skip* tiene la misma
utilidad que el 0 en
aritmética!

La Asignación

La asignación es una de las sentencias más importantes:



El efecto de un asignación es cambiar el valor de una variable:



La Asignación (cont.)

Lo mínimo que requerimos para Q sea verdadero después de S es $Q[x:=E]$.

$$[wp.(x := E).Q \equiv Q(x := E)]$$

Sustitución

Es decir, para que $\{P\}S\{Q\}$ sea verdadero, P debe ser más fuerte que $Q(x:=E)$:

$$\{P\}x := E\{Q\} \equiv [P \Rightarrow Q[x := E]]$$

También utilizaremos la asignación de varias variables:

$x, y, z := x + 1, y * 2, z + 10;$

Ejemplo

Supongamos que queremos demostrar:

$$\{true\}x, y := x + 1, x + 2; \{y = x + 1\}$$

Veamos:

$$[true \Rightarrow (y = x + 1)][x, y := x + 1, x + 2]$$

$$\equiv [Lógica]$$

$$(y = x + 1)[x, y := x + 1, x + 2]$$

$$\equiv [sustitución]$$

$$x + 2 = x + 1 + 1$$

$$\equiv [aritmética]$$

$$true$$

La composición

La composición o secuencia nos permite escribir secuencias de acciones:

$S ; S'$

Primero se ejecuta S y
luego se ejecuta S'

Para probar una composición, tenemos que encontrar un predicado intermedio:

$$\{P\}S; S'\{Q\} \equiv \exists R : \{P\}S\{R\} \wedge \{R\}S'\{Q\}$$

Para calcular el wp de $S;S'$, primero calculamos el wp de S' , y luego la precondición de S

$$wp.(S; S').Q \equiv wp.S.(wp.S'.Q)$$

Ejemplo:

Problemas: $\{x > y\} x := x + 1; y := y + 1 \{x > y\}$

$$x > y \Rightarrow wp.(x := x + 1; y := y + 1).(x > y)$$

$$\equiv [\text{def. wp}]$$

$$x > y \Rightarrow wp.x := x + 1.(wp.y := y + 1.(x > y))$$

$$\equiv [\text{def. wp}]$$

$$x > y \Rightarrow wp.x := x + 1.(x > y + 1)$$

$$\equiv [\text{Aritmética}]$$

$$x > y \Rightarrow (x + 1 > y + 1)$$

También podemos encontrar un predicado intermedio para demostrar la corrección:

$$\{x > y\} x := x + 1; \{x > y + 1\} x := y + 1 \{x > y\}$$

La Sentencia “If”

Una de las sentencias importantes es la alternativa, nos permite definir alternativas de ejecución:

$$\begin{array}{l} \textit{if } B_0 \rightarrow S_0 \\ \square B_1 \rightarrow S_1 \\ \vdots \\ \square B_n \rightarrow S_n \\ \textit{fi} \end{array}$$

guardas

Instrucciones

De todos los comandos S_i que cuya guarda B_i es verdadera, se elige uno de forma no-determinista y se lo ejecuta

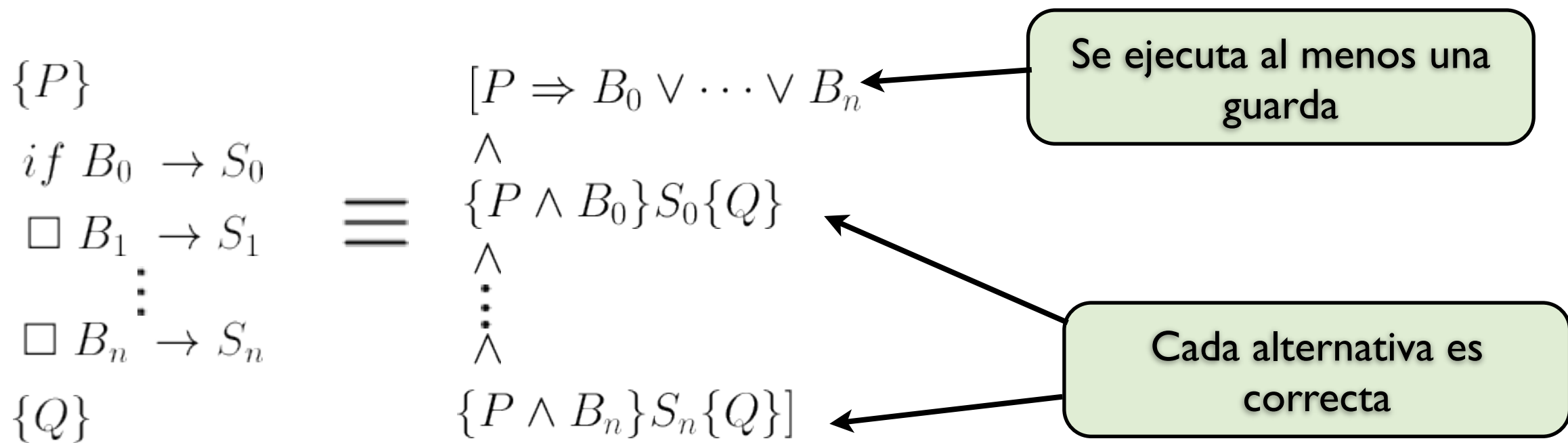
Ejemplo: este programa devuelve $x = \text{cara}$ o $x = \text{seca}$

$$\{True\}$$
$$\textit{if } True \rightarrow x := \textit{cara}$$
$$\square True \rightarrow x := \textit{seca}$$
$$\{x = \textit{cara} \vee x = \textit{seca}\}$$

La Sentencia “If” (cont)

Para probar la corrección:

- Se debe probar que alguna guarda es verdadera.
- Se debe probar que todas las alternativas son correctas.



Probando la corrección del If

Para demostrar un if, tenemos:

I. $[P \Rightarrow B_0 \vee \dots \vee B_n]$

II. $[P \wedge B_i \Rightarrow wp.S_i.Q]$

Para toda rama del If

Por ejemplo:

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

$$if\ x < y \rightarrow x := y$$

$$\square\ x \geq y \rightarrow skip$$

$$\{(x = X \vee x = Y) \wedge x \geq X \wedge x \geq Y\}$$

$$[x = X \wedge y = Y \Rightarrow x < y \wedge x \geq y]$$

$$[x < y \wedge (x = X \wedge y = Y) \Rightarrow wp.(x := y).((x = X \vee x = Y) \wedge x > X \wedge x > Y)]$$

$$[x \geq y \wedge (x = X \wedge y = Y) \Rightarrow wp.skip.((x = X \vee x = Y) \wedge x > X \wedge x > Y)]$$

Derivando Programas

Consideremos el siguiente programa:

$$\{q = a * c \wedge w = c^2\} a, q := a + c, E \{q = a * c\}$$

Encontremos un
valor para E:

Encontramos E!

$$[q = a * c \wedge w = c^2 \Rightarrow wp.(a, q := a + c, E).(q = a * c)]$$

$$\equiv [\text{Def.wp}]$$

$$[q = a * c \wedge w = c^2 \Rightarrow E = (a + c) * c]$$

$$\equiv [\text{Aritmética}]$$

$$[q = a * c \wedge w = c^2 \Rightarrow E = a * c + c^2]$$

$$\equiv [\text{Leibniz}]$$

$$[q = a * c \wedge w = c^2 \Rightarrow E = a * c + w]$$

$$\Leftarrow [\text{Lógica}]$$

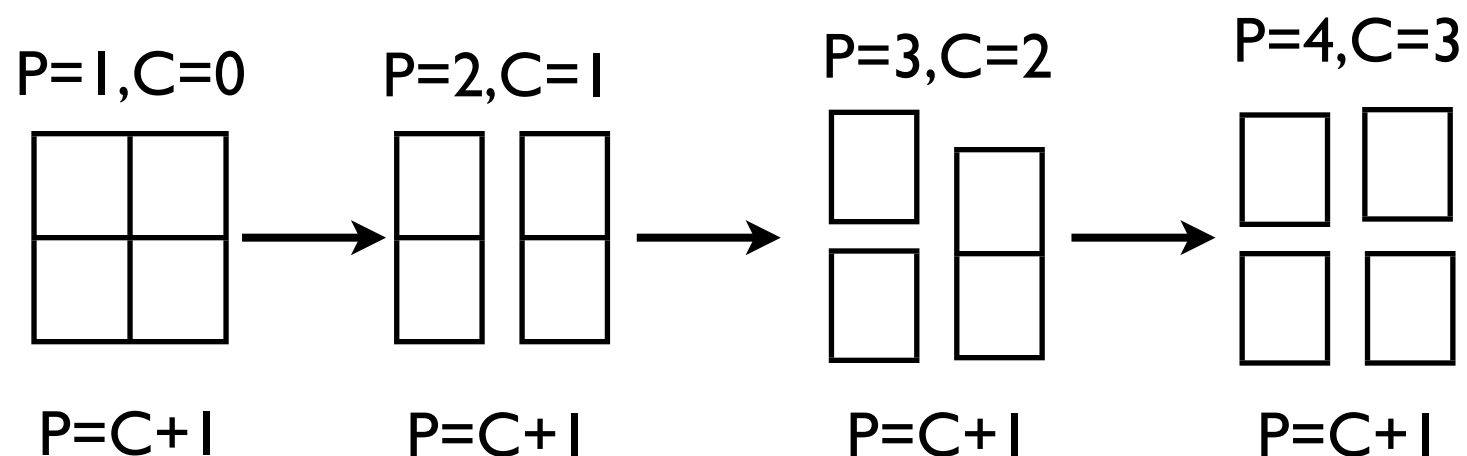
$$[E = a * c + w]$$

Invariantes

Los invariantes nos permiten razonar sobre ciclos:

Ejemplo de la barra de chocolate:

- Un pedazo de chocolate
- N bloques
- Cuantos cortes necesitamos?



$$P = N$$

$$\equiv [Invariante]$$

$$C + 1 = N$$

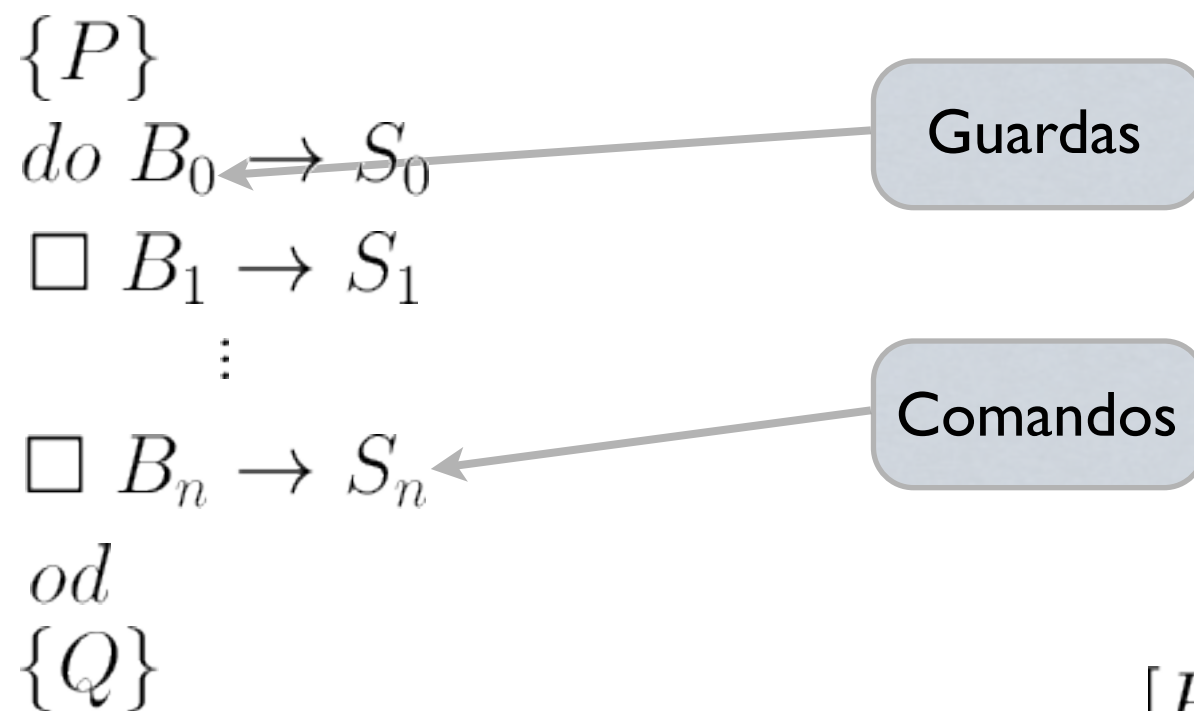
$$\equiv [Aritmética]$$

$$C = N - 1$$

Necesitamos $N-1$ cortes

Repetición

La repetición nos permite ejecutar de forma repetida una acción



Un *do* elige en cada paso una guarda verdadera (de una forma no-determinista), y ejecuta el comando correspondiente. Procede así, hasta que todas las guardas son falsas.

Para demostrar la corrección:

$$\begin{array}{l} [P \wedge \neg B_0 \wedge \cdots \wedge \neg B_n \Rightarrow Q] \\ \wedge \{P \wedge B_0\} S_0 \{Q\} \\ \vdots \\ \wedge \{P \wedge B_n\} S_n \{Q\} \\ \text{y el ciclo termina} \end{array}$$

P se llama invariante

Un Ejemplo

Analicemos el siguiente programa:

$\{n \geq 0 \wedge m \geq 0\}$
 $r, i := 1, 0;$
 $do\ i < m \rightarrow r, i := r * n, i + 1;\ od$
 $\{r = n^m\}$

Inicialización

Postcondición: El programa en r calcula n elevado a m

En cada paso del ciclo tenemos calculado:

Invariante

$r = n^i$
 \wedge
 $i \leq m$

Este predicado es un buen candidato para invariante. Cuando el ciclo termina tenemos $i=m$, lo cual hace verdadera la postcondición

Terminación

La variable i en cada paso del do se acerca más al m .

$$\{P \wedge m - i - 1 = A\} r, i := r * n, i + 1 \{P \wedge m - i - 1 < A\}$$

Además tenemos:

Esto nos garantiza que el ciclo termina, el valor $m-i$ es siempre más chico, pero tiene que parar cuando $m-i=0$

$$i < m \Rightarrow m - i - 1 \geq 0$$

Estas expresiones son llamadas *variantes*, deben cumplir:

$$P \wedge B_i \Rightarrow v \geq 0$$

Durante el ciclo el valor es positivo

$$\{P \wedge v = A\} S_i \{v < A\}$$

En cada paso el valor del variante se decrementa

Teorema del *do*

Para demostrar la corrección de un *do* debemos demostrar los siguientes puntos:

- Inicialización: $\{P\}S\{I\}$

La inicialización hace verdad el invariante

- Postcondición: $I \wedge \neg B_0 \wedge \dots \wedge \neg B_n \Rightarrow Q$

Cuando termina el ciclo vale la postcondición

- Invariante: $\{B_i \wedge I\}S_i\{I\}$

"I" es un invariante

- Variante (a): $I \wedge B_i \Rightarrow v > 0$

Adentro del *do* "v" es siempre positivo

- Variante (b): $\{I \wedge v = A\}S_i\{v < A\}$

Cada paso del *do* decrementa "v"

Ejemplo

Para el caso anterior tenemos que demostrar:

$$1. \{n \geq 0 \wedge m \geq 0\} r, i := 1, 0; \{r = n^i \wedge i \leq m\}$$

$$2. r = n^i \wedge i \leq m \wedge i = m \Rightarrow r = n^m$$

$$3. \{r = n^i \wedge i \leq m \wedge i < m\} r, i := r * n, i + 1; \{r = n^i \wedge i \leq m\}$$

$$4. r = n^i \wedge i \leq m \wedge i < m \Rightarrow m - i > 0$$

$$5. \{r = n^i \wedge i \leq m \wedge m - i = A\} r, i := r * n, i + 1; \{m - i < A\}$$

Y el programa es correcto!

Otro Ejemplo

Suma de un arreglo:

con $N : \text{Int}; a : \text{Array}[0, N) \text{ of } \text{Int};$
var $x, n : \text{Int};$
 $x, n := 0, 0;$
 $\{P : N \geq 0\}$
do $n < N \rightarrow x, n := x + a.n, n + 1;$ *od*
 $\{Q : x = \langle \sum i : 0 \leq i < N : a.i \rangle\}$

En cada paso del ciclo
tenemos en x calculado la
suma $a[0..n]$

Podemos poner como invariante:

$$\{I : (x = \langle \sum i : 0 \leq i < n : a.i \rangle) \wedge n \leq N\}$$