VERIFICACIÓN DE PROGRAMAS CON CÁLCULO DE HOARE

Rodrigo Cardoso
Agosto 2013

Verificar un programa `S` con respecto a una especificación `[Q,R]`, es comprobar que se cumpla

$$\{Q\}\ S\ \{R\}$$

i.e., que el programa `S` sea correcto cra `Q`, `R`.

En general, la estrategia que debe intentarse es fácil, porque solo tiene un camino: dependiendo de la forma de `S` (**skip**, asignación, condicional, secuenciación, iteración) se usa la regla de Hoare correspondiente.

Enseguida se mostrarán ejemplos de cada uno de estos casos. Se supondrá que las instrucciones y las aserciones están sintácticamente bien formadas.

1 VERIFICACIÓN DE SKIP

Para mostrar que

$$\{Q\}\ \textbf{skip}\ \{R\}$$

se tiene:

[Ax skip] $\{Q\}\ \textbf{skip}\ \{R\} \equiv (Q \Rightarrow R)$

Por ejemplo, para mostrar

$$\{Q\colon x > 1\}\ \textbf{skip}\ \{R\colon x > 0\}$$

se observa que:

```
   {Q: x > 1} skip {R: x > 0}
=     ⟨Ax skip⟩
   x>1 ⇒ x>0
=     ⟨Aritmética⟩
   true
```

Eventualmente puede encontrarse que no se escriba **skip** sino que, simplemente, se encuentran dos anotaciones seguidas (sin código entre ellas), así:

```
{Q: x > 1}
{R: x > 0}
```

En este caso, debe entenderse que hay un **skip** entendido.

Más que la demostración formal de que un **skip** se verifique con respecto a una especificación, hay que recordar que la verificación se reduce a probar que la precondición implica la poscondición.

2 VERIFICACIÓN DE ASIGNACIONES

Para mostrar que

$$\{Q\}\ x{:=}\ e\ \{R\}$$

se tiene:

$$[RI\ :=]\quad \frac{Q \Rightarrow R[x:=e]}{\{Q\}\ x:=\ e\ \{R\}}$$

Es decir, la verificación de una asignación se reduce (como la de **skip**) a probar una asignación.

2.1 Ejemplo (asignación simple)

Para mostrar

```
{Q: x = 1} x:= x+4 {R: x > 3}
```

se observa que:

```
   {Q: x = 1} x:= x+4 {R: x > 3}
⇐     ⟨RI :=⟩
   x=1 ⇒ (x>3)[x:= x+4]
=
   x=1 ⇒ x+4>3
=     ⟨Aritmética⟩
   x=1 ⇒ x>-1
=     ⟨Aritmética⟩
   true
```

La regla de inferencia garantiza que, si se cumplen las hipótesis, también vale la conclusión. En este caso, la conclusión es la fórmula de corrección que se desea verificar. Nótese el sentido de la inferencia (⇐) en la demostración.

2.2 Ejemplo (asignación paralela)

Para mostrar

```
{Q: x=A ∧ y=B } x,y:= y,x {R: x=B ∧ y=A}
```

se observa que:

```
   {Q: x=A ∧ y=B } x,y:= y,x {R: x=B ∧ y=A}
⇐     ⟨RI :=⟩
   x=A ∧ y=B ⇒ (x=B ∧ y=A)[x,y:= y,x]
=
   x=A ∧ y=B ⇒ y=B ∧ x=A
=
   true
```

2.2 Ejemplo (asignación de arreglos)[1]

Un tipo de datos que debe existir en un lenguaje de programación como `GCL` es el de los arreglos (matrices, etc.). Un *arreglo* declarado

```
b[p..q]:X
```

debe entenderse como un agregado de datos de tipo `X` que se puede referenciar con expresiones de la forma `b[k]` donde, as u vez, `k` es una expresión que evalúa a un valor entero en el intervalo `p..q`.

Es usual encontrar instrucciones de la forma

```
b[0]:= 19          // Deje el valor 19 en b[0] (el resto del arreglo no cambia)
b[i+1]:= b[j]      // Deje el valor de b[j] en la posición i+1 de b
…
```

[1] Esta sección se incluye solo por mostrar que toda instrucción del lenguaje debe ser verificable. En la práctica es más sencillo, teniendo un cuidado razonable, verificar asignaciones sobre agregados de valores.

Una forma de relacionar estas asignaciones con las que ya se conocen (y con la regla de inferencia de asignación) es entender el arreglo como una variable para que la notación debe entenderse como un cambio parcial. Para explicar esto, si la variable `b` tiene como valor el agregado de los valores que guarda, la notación

$$(b;i:e)$$

se entiende como un arreglo igual a `b`, excepto que en la posición `i` se ha dejado el valor de la expresión `e`.

De este modo, operacionalmente se puede entender que la asignación

$$b[0]:= 19$$

tiene los mismos efectos que

$$b:= (b;0:19)$$

Igualmente, la asignación

$$b[i+1]:= b[j]$$

debe hacer lo mismo que

$$b:= (b;i+1:b[j])$$

Y entonces, la verificación de asignaciones de arreglos es la de una asignación más.

Por ejemplo, supóngase el arreglo `b[0..n-1]:`**`nat`** y se quiere mostrar que

$$\{Q: (\forall i \mid 0 \le i < n-1: b[i]=i^2) \wedge n=6\}\ b[5]:= b[3]+b[n-2]\ \{R: (\forall i \mid 0 \le i < n: b[i]=i^2)\}$$

Es decir:

$$\{Q: (\forall i \mid 0 \le i < n-1: b[i]=i^2) \wedge n=6\}\ b:= (b;5:b[3]+b[n-2])\ \{R: (\forall i \mid 0 \le i < n: b[i]=i^2)\}$$

$\Leftarrow$ $\langle RI :=\rangle$

$$(\forall i \mid 0 \le i < n-1: b[i]=i^2) \wedge n=6 \ \Rightarrow\ (\forall i \mid 0 \le i < n: b[i]=i^2)[b:= (b;5:b[3]+b[n-2])]$$

=

$$(\forall i \mid 0 \le i < n-1: b[i]=i^2) \wedge n=6 \ \Rightarrow\ (\forall i \mid 0 \le i < n: (b;5:b[3]+b[n-2])[i]=i^2)$$

Esta afirmación se puede mostrar con una prueba por hipótesis (teorema de la deducción)

Hip: $(\forall i \mid 0 \le i < n-1: b[i]=i^2)$, $n=6$ // Mostrar: $(\forall i \mid 0 \le i < n: (b;5:b[3]+b[n-2])[i]=i^2)$

$(\forall i \mid 0 \le i < n: (b;5:b[3]+b[n-2])[i]=i^2)$

= $\langle$Partir rango a la derecha$\rangle$

$(\forall i \mid 0 \le i < n-1: (b;5:b[3]+b[n-2])[i]=i^2) \wedge ((b;5:b[3]+b[n-2])[i]=i^2)[i:=n-1]$

=

$(\forall i \mid 0 \le i < n-1: (b;5:b[3]+b[n-2])[i]=i^2)$
$\wedge\ ((b;5:b[3]+b[n-2])[n-1]= (n-1)^2)$

= $\langle$Hip: n=6; Aritmética$\rangle$

$(\forall i \mid 0 \le i < 5: (b;5:b[3]+b[4])[i]=i^2)) \wedge ((b;5:b[3]+b[4])[5]= 5^2)$

= $\langle$i<5 en la cuantificación: solo en la pos. 5$\rangle$

$(\forall i \mid 0 \le i < 5: b[i]=i^2) \wedge ((b;5:b[3]+b[4])[5]= 5^2)$

= $\langle$(b;5:b[3]+b[4])[5] = b[3]+b[4]$\rangle$

$(\forall i \mid 0 \le i < 5: b[i]=i^2) \wedge b[3]+b[4]= 5^2)$

= $\langle$Hip: $(\forall i \mid 0 \le i < n-1: b[i]=i^2)$, n=6$\rangle$

$b[3]+b[4]= 5^2)$

= $\langle$Hip: $b[3]=3^2 \wedge b[4]=4^2\rangle$

true

3 VERIFICACIÓN DE CONDICIONALES

Para verificar una instrucción condicional de la forma

$$\{Q\}\ \text{IF}\ \{R\}$$

con

$$\text{IF: } \textbf{if } B1 \rightarrow S1\ [] \ \ldots\ [] \ Br \rightarrow Sr\ \textbf{fi}$$

debe recordarse la regla de inferencia correspondiente:

$$[\text{RI IF}]: \quad \frac{Q \Rightarrow BB,\ \text{Para } i=1,\ldots,r: \{Q \wedge Bi\}\ Si\ \{R\}}{\{Q\}\ \text{IF}\ \{R\}}$$

donde

$$BB \equiv (\exists i |\ 1 \leq i \leq r: Bi)$$

3.1 Ejemplo (valor absoluto)

Para mostrar la corrección de

$$\begin{array}{l} \{Q: x = A\} \\ \textbf{if } x<0 \rightarrow x := -x \\ [] \ x \geq 0 \rightarrow \textbf{skip} \\ \textbf{fi} \\ \{R: x = |A|\} \end{array}$$

se procede así:

$$\begin{array}{l} BB \equiv x<0 \vee x \geq 0 \\ \quad \equiv \text{true} \end{array}$$

Se muestran:

(i) $Q \Rightarrow BB$

(ii) $\{Q \wedge x<0\}\ x := -x\ \{R\}$

(iii) $\{Q \wedge x \geq 0\}\ x := -x\ \{R\}$

(i)

$$\begin{array}{ll} & Q \Rightarrow BB \\ = & \\ & x = A \Rightarrow \text{true} \\ = & \\ & \text{true} \end{array}$$

(ii)

$$\begin{array}{ll} & \{x=A \wedge x<0\}\ x := -x\ \{R\} \\ = & \quad \langle Ax :=\rangle \\ & x=A \wedge x<0 \Rightarrow (x = |A|)[x := -x] \\ = & \\ & x=A \wedge x<0 \Rightarrow -x = |A| \\ = & \quad \langle \text{Aritmética} \rangle \\ & \text{true} \end{array}$$

(iii)

$$\begin{array}{ll} & \{x=A \wedge x \geq 0\}\ \textbf{skip}\ \{R\} \\ = & \quad \langle Ax\ \text{skip}\rangle \end{array}$$

```
   x=A ∧ x≥0 ⇒ x = |A|
=     ⟨Aritmética⟩
   true
```

Este ejemplo muestra una situación muy común, en la que la disyunción de las guardas es `true`, debido a que se consideran todos los casos posibles. Esto es usual cuando se cuenta con una instrucción **if-then-else**. En `GCL`, esto se puede simular con una condicional de dos guardas donde cada una equivale a la negación de la otra, i.e.,

```
ITE:  if  B → S1
      [] ¬B → S2
      fi
```

Y es claro que, siendo `true` la disyunción de las guardas, `BB` siempre es implicada por la precondición. Es decir, en este caso la verificación puede obviar esta comprobación.

4 VERIFICACIÓN DE SECUENCIACIONES

Para verificar la corrección de

```
{Q} S1; S2 {R}
```

se cuenta con la regla

$$[RI\ ;]: \quad \frac{\{Q\}\ S1\ \{R1\},\ \{R1\}\ S2\ \{R\}}{\{Q\}\ S1;\ S2\ \{R\}}$$

La aserción `R1` describe el estado intermedio que alcanza la ejecución de `S1`, la primera parte de la instrucción compuesta. A partir de este punto, la segunda parte, `S2`, debe ejecutarse y terminar en `R`.

La aplicación de esta regla exige de quien realiza la verificación el ser capaz de inventar o deducir una tal aserción `R1`. En realidad, se supone que quien verifica es el mismo desarrollador[2], de modo que él debería saber qué aserción `R1` utilizar.

4.1 Ejemplo (intercambio secuencial de variables)

El siguiente ejemplo debería resultar conocido para cualquier programador:

```
{Q: x=A ∧ y=B}
   t:= x; x:= y; y:= t;
{R: x=B ∧ y=A}
```

La verificación formal requiere aserciones intermedias que describan lo que sucede después de cada asignación. Nótese que lo que se da como aserción es, simplemente, lo que se quiere decir que la asignación haga:

```
{Q: x=A ∧ y=B}
      t:= x;
{Q1: t=A ∧ x=A ∧ y=B}
      x:= y;
{Q2: t=A ∧ x=B ∧ y=B}
      y:= t;
{R: x=B ∧ y=A}
```

[2] Un desarrollador de software debería verificar sus programas, en la situación ideal. Además, debería documentarlos con aserciones intermedias en los casos en que se requiriera.

En este punto, el uso de la regla `RI;` consiste en mostrar la corrección de cada una de las asignaciones con respecto a su correspondiente especificación, i.e., se deben mostrar (¡ejercicio!):

```
(i)   {Q } t:= x {Q1}
(ii)  {Q1} x:= y {Q2}
(iii) {Q2} y:= t {R }
```

5 VERIFICACIÓN DE ITERACIONES

Como en el caso de la verificación de secuencias de instrucciones, la verificación de instrucciones iterativas requiere de quien programa el documentar el código para hacer posible la utilización de la regla correspondiente[3]:

$$\textbf{[RI DO]:}\quad \frac{\{Q\}\text{INIC}\{P\},\ P\wedge\neg BB\Rightarrow R,\ \text{Para } i=1,\ldots,n\text{: } \{P\wedge Bi\}Si\{P\}),\ \text{Terminación}}{\{Q\}\ \text{INIC; } \{\text{inv } P\}\ \text{DO } \{R\}}$$

También conocida como el *Teorema del DO*, la regla se puede expresar de la siguiente forma, en donde se ha incluido una manera formal de comprobar la terminación, con la ayuda de una función *cota*

$$t: D \rightarrow \textbf{int}$$

con la intención de estimar (por encima) el número de iteraciones que hacen falta para acabar la ejecución.

5.1 Teorema del DO

Para verificar:

```
{Q}
INIC;
{inv P}
{cota t}
do BB → IF od
{R}
```

basta chequear las afirmaciones:

1	`P` *vale antes*, i.e.,	$\{Q\}\ \text{INIC}\ \{P\}$
2	`P` *sirve*, i.e.,	$P \wedge \neg BB \Rightarrow R$
3	`P` *invariante*, i.e.,	$\{P \wedge BB\}\ \text{IF}\ \{P\}$
4	Hay *terminación*, i.e.,	$\{P \wedge BB \wedge t=t_0\}\ \text{IF}\ \{t<t_0\}$
		$P \wedge BB \Rightarrow t>0$

La aserción `P`, el *invariante*, es la ficha principal del juego de la verificación de una instrucción iterativa. Nótese cómo aparece en todas las condiciones que se deben chequear.

Para entender por qué los nombres de las condiciones de chequeo:

`P` *vale antes*: garantiza que el invariante vale justo antes de empezar a iterar. Desde la precondición puede requerirse una *inicialización*, i.e., una instrucción fácil de entender y de ejecutar, que lleva al estado a satisfacer `P`.

`P` *invariante*: una vez que se llega a iterar, si esto es posible (porque vale `BB`), y se comienza la iteración en un estado que cumple `P`, después de ejecutar un ciclo se llega a un estado que también cumple `P`. Si ya se

[3] De hecho, aunque se podría pensar en desarrollar técnicas para descubrir esta documentación, se sabe que esto es teóricamente imposible.

sabe que P vale antes (al empezar), este chequeo garantiza que P se cumple siempre, al tratar de iterar. De aquí su nombre de *invariante*. Un ciclo puede tener muchos invariantes que cumplan esta definición. En particular, obsérvese que true siempre es un posible invariante.

P *sirve*: De los invariantes que puede tener el DO, solo algunos servirán para mostrar la poscondición, en el momento de terminar (cuando no valga BB). Por eso el nombre del chequeo. Aunque los únicos que sirven son invariantes, es más simple probar esta condición primero que la invariancia misma; si no se puede probar que P sirva, no hay que esforzarse en probar cosas más complejas sobre P.

La *terminación* se garantiza si se cumplen los chequeos indicados para la función cota. El primer chequeo afirma que, si se efectúa una iteración la cota rebaja efectivamente. El segundo chequeo afirma que si hay alguna guarda que valga, la cota debe ser positiva. Así, sabiendo que la cota toma valores enteros, si hay que iterar se debe comenzar en un valor positivo. Y cada iteración debe bajar la cota de manera estricta. En otras palabras, el valor de la cota será, después de un número finito de iteraciones, menor o igual a 0. En este momento no puede valer BB, y la instrucción debe terminar.

En la práctica es usual mostrar la terminación de manera informal. Por ejemplo, si el ciclo es una especie de **for** (como en java o en C), es usual tener un contador que recorre un intervalo con el que se estima fácilmente una cota. Entonces se justifica la terminación de manera informal y se usa la regla de inferencia sin chequear las partes de terminación.

5.2 Ejemplo (multiplicación binaria)

El siguiente algoritmo es útil para multiplicar fácilmente números expresados en notación binaria. En binario es fácil detectar si un número es par (termina en 0), dividirlo por 2 cuando es par (un *shift* a la derecha), restar 1 cuando es impar (hacer 0 el bit de las unidades), y duplicar un valor (un *shift* a la izquierda). El algoritmo supone además ya disponible un algoritmo de suma.

Se quiere verificar que el siguiente algoritmo sirve para multiplicar a*b, con b≥0:

```
{Q: b≥0}

x,y,z:= a,b,0;

{Inv P: y≥0 ∧ z+x*y = a*b}
{cota t: y}

do y>0 ∧ par.y          →     x,y:= x+x,y÷2
[] ¬par.y               →     z,y:= z+x,y-1
od

{R: z = a*b}
```

[1] P vale antes

```
   {Q} x,y,z:= a,b,0 {P}
=      ⟨Ax :=⟩
   Q ⇒ P[x,y,z:= a,b,0]
=
   b≥0 ⇒ b≥0 ∧ 0+a*b = a*b
   (1)     (2)         (3)
=      ⟨1 ⇒ 2: p⇒p; 3: aritmética⟩
   true
```

[2] P sirve

```
   BB
=
  (y>0 ∧ par.y) ∨ ¬par.y
=     ⟨absorción⟩
   y>0 ∨ ¬par.y
```

Por tanto:

```
   ¬BB
=     ⟨DeMorgan⟩
   y≤0 ∧ par.y
=     ⟨y: nat⟩
   y=0

Hip: (P, ¬BB) y≥0, z+x*y = a*b, y=0
   true
=     ⟨Hip: z+x*y = a*b⟩
   z+x*y = a*b
=     ⟨Hip: y=0; u+0=u⟩
   z = a*b
```

[3] P invariante

[a]

```
   {P ∧ y>0 ∧ par.y} x,y:= x+x,y÷2 {P}
=
   P ∧ y>0 ∧ par.y ⇒ P[x,y:= x+x,y÷2]
=
   y≥0 ∧ z+x*y = a*b ∧ y>0 ∧ par.y ⇒ y÷2≥0 ∧ z+(x+x)*(y÷2) = a*b
      (1)         (2)       (3)     (4)       (5)                  (6)
=     ⟨1 ⇒ 5: propiedades de ÷
          (4 ⇒ y÷2 = y/2; x+x = 2*x; Cancelacion-*) ⇒ 6 ⟩
   true
```

[b]

```
   {P ∧ ¬par.y} z,y:= z+x,y-1 {P}
=
   P ∧ ¬par.y ⇒ P[z,y:= z+x,y-1]
=
   y≥0 ∧ z+x*y = a*b ∧ ¬par.y ⇒ y-1≥0 ∧ z+x+x*(y-1) = a*b
      (1)          (2)       (3)        (4)                (5)
=     ⟨1 ∧ 3 ⇒ 4: aritmética;
          2 ⇒ 5 : aritmética⟩
   true
```

[4] Terminación

Informal: la cota `y` disminuye efectivamente en cada iteración. En el primer caso debe notarse que `y` es par y positivo, de modo que `y/2<y`.

Formalmente:

[a]

```
   {P ∧ y>0 ∧ par.y ∧ y=y0} x,y:= x+x,y÷2 {y<y0}
=
   P ∧ y>0 ∧ par.y ∧ y=y0 ⇒ (y<y0)[x,y:= x+x,y÷2]
=
   y≥0 ∧ z+x*y = a*b ∧ y>0 ∧ par.y ∧ y=y0 ⇒ y÷2<y0
      (1)        (2)      (3)    (4)    (5)     (6)
=     ⟨3 ∧ 4 ∧ 5 ⇒ 6: propiedades de ÷⟩
   true
```

[b]

```
   {P ∧ ¬par.y ∧ y=y0} z,y:= z+x,y-1 {y<y0}
=
   P ∧ ¬par.y ∧ y=y0 ⇒ (y<y0)[z,y:= z+x,y-1]
=
   y≥0 ∧ z+x*y = a*b ∧ ¬par.y ∧ y=y0 ⇒ y-1<y0
      (1)        (2)        (3)    (4)    (5)
=     ⟨4 ⇒ 5: aritmética⟩
   true
```

5.3 Ejemplo (*Coffee can problem*)

Considérese el problema (cf. [Gri1983], [Car1993]):

"Una bolsa tiene `B` bolas blancas y `N` negras, y no está vacía. El siguiente proceso se repite siempre que se pueda:

- extraer dos bolas
- si son del mismo color, se tiran, pero se entra una negra a la bolsa
- si son de diferente color, se devuelve la blanca y se tira la negra

Se quiere saber si el proceso termina y, en caso de que así sea, cómo es el estado final de la bolsa."

La extracción de dos bolas en cada iteración del proceso no se precisa. Solo se requiere que se extraigan dos bolas, de manera que esto se puede entender que se hace de manera arbitraria, i.e., no determinística.

Se puede probar que el proceso termina con una bola y ésta será negra, si y solo si el número inicial de bolas blancas es par.

El siguiente algoritmo simula el proceso. Obsérvese cómo la documentación (`P`, `t` y lo que se afirma) aclaran el proceso y la explicación de la argumentación:

```
{Pre Q: 0≤b=B ∧ 0≤n=N ∧ b+n>0}
{Inv P: 0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B}
{Cota t: b+n}

      do b≥2 ∧ n≥0      →    b,n:= b-2,n+1     // sacar 2 blancas
      [] b≥0 ∧ n≥2      →    n:= n-1           // sacar 2 negras
      [] b≥1 ∧ n≥1      →    n:= n-1           // sacar una blanca, una negra
      od

{Pos R: (par.B ∧ b=0 ∧ n=1) ∨ (¬par.B ∧ b=1 ∧ n=0)}
```

[1] P vale antes:

```
   {Q: 0≤b=B ∧ 0≤n=N ∧ b+n>0}{P: 0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B}
=
   0≤b=B ∧ 0≤n=N ∧ b+n>0  ⇒  0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B
     (1)      (2)     (3)       (4)    (5)    (6)         (7)
=     ⟨ (1)⇒(4), (2)⇒(5), (3)⇒(6), (1)⇒(7) ⟩
   true
```

[2] P sirve:

```
   BB
=
   (b≥2 ∧ n≥0) ∨ (b≥0 ∧ n≥2) ∨ (b≥1 ∧ n≥1)
=     ⟨Casos: b≥2 ∧ n≥0; b≥0 ∧ n≥2; b≥1 ∧ n≥1 ⟩
   b+n≥2 ∧ b≥0 ∧ n≥0
```

Entonces:

```
   P ∧ ¬BB
=
   0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B ∧ ¬(b+n≥2 ∧ b≥0 ∧ n≥0)
=     ⟨ DeMorgan ⟩
   0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B ∧ (b+n<2 ∨ b<0 ∨ n<0)
=     ⟨ Absorción: 0≤b ∧ 0≤n ∧ (b+n<2 ∨ b<0 ∨ n<0) ≡ b+n<2 ∧ 0≤b ∧ 0≤n ⟩
   0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B ∧ b+n<2
=     ⟨ … ⟩
   (b=0 ∧ n=1 ∧ par.B) ∨ (b=1 ∧ n=0 ∧ ¬par.B)
```

[3] P invariante:

Guarda 1:

```
   {P ∧ b≥2 ∧ n≥0} b,n:= b-2,n+1 {P}
=
   P ∧ b≥2 ∧ n≥0 ⇒ P[b,n:= b-2,n+1]
=
   0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B ∧ b≥2 ∧ n≥0
            ⇒ (0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B)[b,n:= b-2,n+1]
=
   0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B ∧ b≥2 ∧ n≥0
            ⇒ 0≤b-2 ∧ 0≤n+1 ∧ b-2+n+1>0 ∧ par(b-2)≡par.B
=     ⟨ … ⟩
   true
```

Guarda 2:

```
   {P ∧ b≥0 ∧ n≥2} n:= n-1 {P}
=
   P ∧ b≥0 ∧ n≥2 ⇒ P[n:= n-1]
=
   0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B ∧ b≥0 ∧ n≥2
            ⇒ (0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B)[n:= n-1]
=
   0≤b ∧ 0≤n ∧ b+n>0 ∧ par.b≡par.B ∧ b≥0 ∧ n≥2
```

$\Rightarrow\ 0\le b \wedge 0\le n-1 \wedge b+n-1>0 \wedge par.b\equiv par.B$

$=\quad \langle \ldots \rangle$

$true$

Guarda 3:

$\{P \wedge b\ge 1 \wedge n\ge 1\}\ n:= n-1\ \{P\}$

$=$

$P \wedge b\ge 1 \wedge n\ge 1 \Rightarrow P[n:= n-1]$

$=$

$0\le b \wedge 0\le n \wedge b+n>0 \wedge par.b\equiv par.B \wedge b\ge 1 \wedge n\ge 1$

$\Rightarrow\ (0\le b \wedge 0\le n \wedge b+n>0 \wedge par.b\equiv par.B)[n:= n-1]$

$=$

$0\le b \wedge 0\le n \wedge b+n>0 \wedge par.b\equiv par.B \wedge b\ge 1 \wedge n\ge 1$

$\Rightarrow\ 0\le b \wedge 0\le n-1 \wedge b+n-1>0 \wedge par.b\equiv par.B$

$=\quad \langle \ldots \rangle$

$true$

[4] Terminación:

Un argumento informal sencillo: en cada iteración, el número total de bolas baja en 1. Se termina porque no se pueden sacar dos bolas, es decir, dejando solo una.

Formalmente:

La cota es positiva si se puede iterar: $P \wedge BB \Rightarrow t>0$

$P \wedge BB \Rightarrow b+n>0$

$=$

$0\le b \wedge 0\le n \wedge b+n>0 \wedge par.b\equiv par.B \wedge b+n\ge 2 \wedge b\ge 0 \wedge n\ge 0$

$\Rightarrow\quad \langle \ldots \rangle$

$b+n>0$

La cota baja en cada iteración:

Guarda 1:

$\{P \wedge b\ge 2 \wedge n\ge 0 \wedge b+n=t_0\}\ b,n:= b-2,n+1\ \{b+n<t_0\}$

$=$

$P \wedge b\ge 2 \wedge n\ge 0 \wedge b+n=t_0 \Rightarrow (b+n<t_0)[b,n:= b-2,n+1]$

$=$

$P \wedge b\ge 2 \wedge n\ge 0 \wedge b+n=t_0 \Rightarrow b-2+n+1<t_0$

$=$

$true$

Guarda 2:

$\{P \wedge b\ge 0 \wedge n\ge 2 \wedge b+n=t_0\}\ n:= n-1\ \{b+n<t_0\}$

$=$

$P \wedge b\ge 0 \wedge n\ge 2 \wedge b+n=t_0 \Rightarrow (b+n<t_0)[n:= n-1]$

$=$

$P \wedge b\ge 0 \wedge n\ge 2 \wedge b+n=t_0 \Rightarrow b+n-1<t_0$

$=$

$true$

Guarda 3:

$\{P \wedge b \geq 1 \wedge n \geq 1 \wedge b+n=t_0\}\ n := n-1\ \{b+n<t_0\}$

=

$P \wedge b \geq 1 \wedge n \geq 1 \wedge b+n=t_0 \Rightarrow (b+n<t_0)[n := n-1]$

=

$P \wedge b \geq 1 \wedge n \geq 1 \wedge b+n=t_0 \Rightarrow b+n-1<t_0$

=

$true$