GCL: SEMÁNTICA OPERACIONAL Y AXIOMÁTICA

Rodrigo Cardoso
Enero de 2015

GCL *(Guarded Commands Language)* es un lenguaje algorítmico sencillo que tiene las estructuras de control básicas de cualquier lenguaje de programación. Una descripción detallada de GCL se encuentra en [Car1993].

Para estudiar algoritmos es conveniente disponer de un lenguaje en el que se puedan describir los algoritmos que sea sencillo y fácil de implementar en lenguajes de programación comerciales. La sintaxis del lenguaje algorítmico es importante, pero no se atiende demasiado a ella, si se piensa que al traducir a un lenguaje de programación real, el chequeo sintáctico se realiza mecánicamente. En cambio, la semántica debe ser bien entendida, pues la traducción entre los dos formalismos es correcta precisamente cuando la semántica se preserva.

1 SINTAXIS GCL

La siguiente gramática describe la forma de los programas GCL:

```
⟨programa GCL⟩ ::= ⌈ ⟨declaraciones⟩ ⌈ ⟨cuerpo⟩ ⌋
⟨declaraciones⟩::= "declaraciones de variables y tipos"
⟨cuerpo⟩       ::= ⟨instrucción⟩
⟨instrucción⟩  ::= ⟨instrucción⟩; ⟨instrucción⟩
                 | ⟨asignación⟩
                 | ⟨condicional⟩
                 | ⟨iteración⟩
                 | ⟨procedimiento⟩
                 | skip
                 | abort
```

No se dará una descripción detallada de la sintaxis de las instrucciones, pero se explicará (informalmente) la forma de cada una de ellas. Una descripción más completa del lenguaje incluye expresiones para procedimientos y funciones.

1.1 Asignaciones

Son de la forma

$x := e$

donde x es una variable y e es una expresión del mismo tipo que x. Más generalmente, se puede asignar una lista de variables con una lista de expresiones de igual tipo (y tamaño!):

$x_1, \ldots, x_n := e_1, \ldots, e_n$

También se debe considerar la posibilidad de asignar a partes de un arreglo con instrucciones como

$b[i] := e$

1.2 Condicionales

Son de la forma

if $B_1 \rightarrow S_1$
$[]\ B_2 \rightarrow S_2$

…
[] $B_n \rightarrow S_n$
fi

donde $0 \leq n$. Los B_i's son predicados y los S_i's son instrucciones GCL.

1.3 Iteraciones

Son de la forma

do $B_1 \rightarrow S_1$
[] $B_2 \rightarrow S_2$
…
[] $B_n \rightarrow S_n$
od

donde $0 \leq n$. Los B_i's son predicados y los S_i's son instrucciones GCL.

2 SEMÁNTICA OPERACIONAL

Una manera de entender lo que hace un programa escrito en un lenguaje dado (como GCL) es explicar lo que sucede a un autómata o *interpretador* que lo ejecuta. Es decir se establecen los estados del autómata y se explican las transiciones entre ellos que resultan de la ejecución.

Un interpretador para un programa GCL tiene como *espacio de estados* las tuplas de valores de las variables del programa. Así, si el programa tiene variables $x_1, \ldots, x_n$, de tipos $T_1, \ldots, T_n$, un *estado* es una tupla de valores $\langle v_1, \ldots v_n \rangle \in T_1 \times \ldots \times T_n$. En otras palabras, $E = T_1 \times \ldots \times T_n$ es el *espacio de estados* y cada uno de ellos se puede representar por el vector de variables $x = \langle x_1, \ldots, x_n \rangle$.

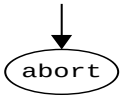
El efecto de una instrucción sobre el estado se explicará mediante un fragmento de un *diagrama de flujo* que describe los cambios de estado en el autómata interpretador[1]. A todas las instrucciones que denotan un comportamiento normal tienen un diagrama con una entrada y una salida. Eso permite componerlas de manera única y sencilla, así como asignar una semántica a todo el programa.

2.1 Instrucción `skip`

Es la instrucción más simple de GCL: no hace nada. A ella corresponde el diagrama de flujo que solo tiene un arco:

2.2 Instrucción abort

Es una instrucción que poco se usa en GCL, pero que se incluye por razones teóricas. Se quiere hacer corrresponder a un comportamiento reconocidamente anormal (terminación anormal). A ella corresponde el diagrama de flujo que tiene un nodo etiquetado **`abort`**, del que no salen arcos:



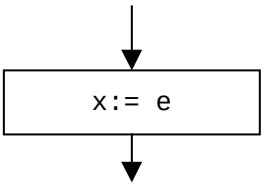
[1] En [Car1993] se explica con detalle el funcionamiento de los diagramas de flujo y se demuestra su equivalencia expresiva con GCL.

2.3 Asignaciones

A una asignación

```
x:= e
```

corresponde un diagrama de flujo:

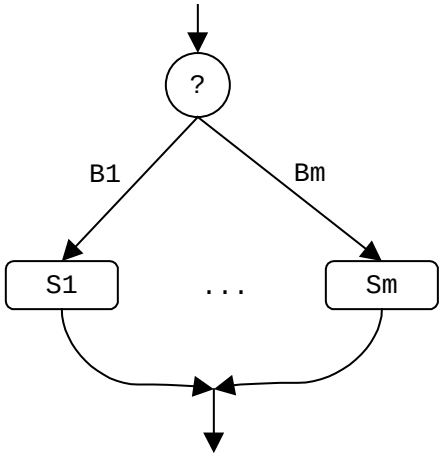


2.4 Condicionales

A una instrucción

```
IF:  if B1 → S1
     [] B2 → S2
     …
     [] Bn → Sn
     fi
```

corresponde un diagrama de flujo:

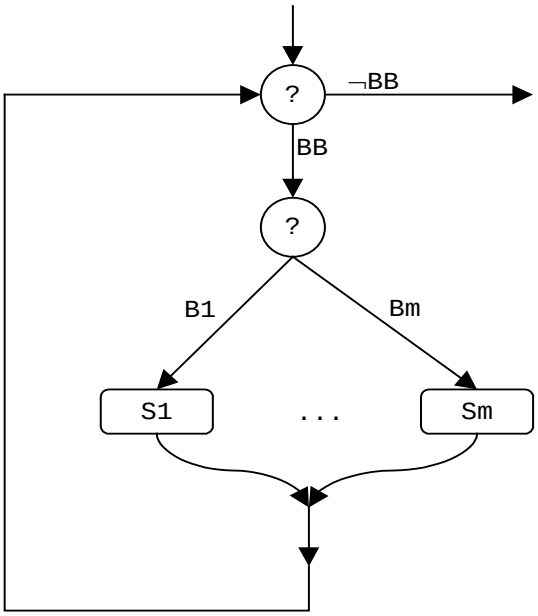


2.5 Iteraciones

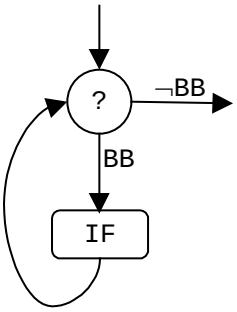
A una instrucción de la forma:

```
DO:  do B1 → S1
     [] B2 → S2
     …
     [] Bn → Sn
     od
```

corresponde un diagrama de flujo:



O, equivalentemente:



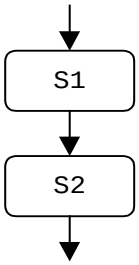
En los dos diagramas, `BB` ≡ `(`∨`k | 1`≤`k`≤`n : Bk)`.

2.6 Composición de instrucciones

La instrucción

```
S1; S2
```

tiene como semántica operacional el diagrama



donde los subdiagramas etiquetados con `S1` y `S2` corrresponden a las instrucciones respectivas.

2.7 Diagramas de flujo de programas

De acuerdo con las reglas de sintaxis, un programa GCL tiene la forma:

⟨programa GCL⟩ ::= ⌈ ⟨declaraciones⟩ ⌈ ⟨instrucción⟩ ⌋

Las ⟨declaraciones⟩ determinan el *espacio de estado* del programa, como el producto cartesiano de los dominios de las variables involucradas. Las instrucciones se refieren al estado, de manera parcial, cuando mencionan las variables; en particular, las instrucciones de asignación pueden cambiar el estado.

En las subsecciones anteriores se ha insistido en cómo entender cada instrucción como un fragmento de diagrama de flujo. En otras palabras, cada instrucción se *traduce* en un fragmento de diagrama de flujo. Es importante notar que, en la mayoría de los casos, las traducciones corresponden a grafos con un nodo de entrada y uno de salida.

En general, un *diagrama de flujo* es una cuádrupla

F = (G,D,L,**x**)

donde:

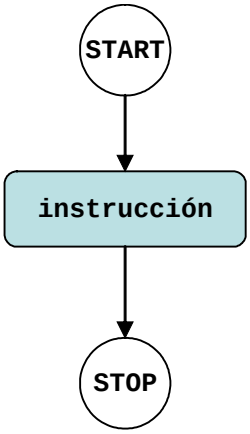
G : es un grafo dirigido, llamado el *grafo de flujo*. Los nodos de G representan acciones o decisiones y son de alguna de las formas:

D : es el espacio de estados.

L : es un conjunto de etiquetas que identifican unívocamente los nodos del grafo G.
Para un nodo n del grafo G, et(n) es la etiqueta de ese nodo.

x : es el vector de estados.

El grafo de flujo correspondiente a un programa GCL es de la forma:



donde el nodo sombreado representa el fragmento de grafo de flujo al que se traduce la instrucción que constituye el cuerpo del programa.

2.8 Ejecución de diagramas de flujo

Considérese un diagrama de flujo F = (G,D,L,**x**).

El conjunto de *situaciones* se define como $L \times D$. Una *situación* es, entonces, una pareja $\langle e, v \rangle$, donde $e \in L$ y $v \in D$.

Una *ejecución de* F a partir de un estado $v \in D$ es una sucesión de situaciones de la forma

$$\langle \langle et(START), v \rangle, \langle e_1, v_1 \rangle, \langle e_2, v_2 \rangle, \ldots \rangle$$

que puede ser finita o infinita. Cuando es finita, se dice que la ejecución *termina*. Un programa *termina* si toda ejecución termina. La terminación es *normal* si el la última situación en la sucesión tiene como etiqueta la de un nodo de parada (**STOP**). En otro caso hay *terminación anormal* (**ABORT**) o *no-terminación*.

La sucesión se define de manera inductiva. A partir de una situación $\langle e_k, v_k \rangle$, se define la situación siguiente $\langle e_{k+1}, v_{k+1} \rangle$ o bien, si es el caso, se establece que no hay sucesor definido (i.e., hay terminación). La definición de la siguiente situación depende del nodo que etiqueta e_k:

Caso: e_k es un nodo **START**:
El nodo **START** debe estar conectado con el nodo por el que empieza la traducción del cuerpo del programa. Sea e la etiqueta de este nodo. Entonces: $\langle e_{k+1}, v_{k+1} \rangle = \langle e, v_k \rangle$.

Caso: e_k es un nodo **STOP**:
El nodo **STOP** no tiene sucesores. En este caso la situación sucesor no se define y la sucesión termina normalmente.

Caso: e_k es un nodo **ABORT**:
El nodo **ABORT** no tiene sucesores. En este caso la situación sucesor no se define y la sucesión termina anormalmente.

Caso: e_k es un nodo de decisión (etiquetado **?**):
Un nodo de decisión debe tener 0 o más sucesores en el grafo de flujo. Los arcos que conectan los sucesores están etiquetados con predicados $B_1, B_2, \ldots, B_r$, para cierto $r \geq 0$. Si en el estado v_k vale alguno de los predicado B_i, entonces se puede tener que $\langle e_{k+1}, v_{k+1} \rangle = \langle e', v_k \rangle$, siendo e' la etiqueta del nodo que conecta al nodo etiquetado por e con el arco etiquetado con B_i.
Puede haber más de un predicado B_i que valga en el estado v ; en caso de haber más de uno se elige no determinísticamente uno de ellos. Si $r=0$ o si ningún predicado B_i vale, la sucesión termina anormalmente (lo cual equivale a abortar).

Caso: e_k es un nodo de asignación (**x**:= f(**x**)).
Este nodo debe estar conectado con un nodo del grafo (y sólo uno), etiquetado –digamos- con e'. En este caso, debe cumplirse que $\langle e_{k+1}, v_{k+1} \rangle = \langle e', f(v_k) \rangle$.

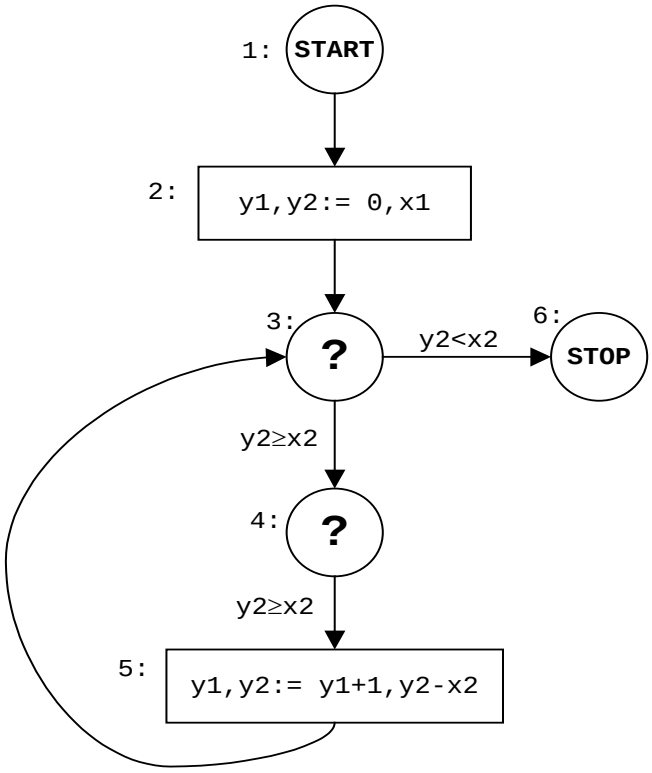
De esta manera, es claro en qué consiste ejecutar un programa GCL: se debe traducir el programa a diagrama de flujo y considerar las posibles ejecuciones del mismo.

Ejemplo:

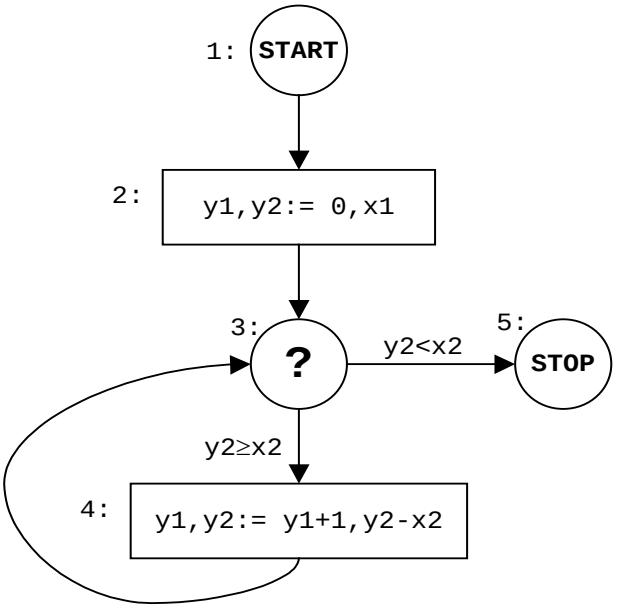
Sea S el programa GCL:

```
⌈ var x1: nat init A;    /* A es un número natural */
  var x2: nat init B;    /* B es un número natural */
  var y1,y2: nat
⌈ y1,y2:= 0,x1;
  do y2 ≥ x2 → y1,y2:= y1+1,y2-x2 od
⌋
```

S se traduce a un diagrama de flujo `F = (G,D,L,x)` así:
`x = (x1,x2,y1,y2)`
`D = nat × nat × nat × nat`
`L = 1..6`
`G:`



La traducción se ha definido de manera que, en este caso, los nodos 3 y 4 hacen la misma pregunta. Uno esperaría que este diagrama "significara lo mismo" que el "reducido":
`L' = 1..5`
`G':`



Una ejecución de S parte de un estado v_0, i.e., una cuádrupla números naturales:

$$(A,B,\perp,\perp)$$

El valor "⊥" se puede leer como "indefinido" o "indeterminado".

a Supóngase $v_a = (3,10,\perp,\perp)$. Solo es posible generar una ejecución de S desde v_a:

```
〈  (1, (3,10,⊥,⊥),
   (2, (3,10,⊥,⊥),
   (3, (3,10,0,3),
   (6, (3,10,0,3)  〉
```

b Supóngase $v_b = (10,3,\perp,\perp)$. Solo es posible generar una ejecución de S desde v_b:

```
〈  (1, (10,3,⊥,⊥),
   (2, (10,3,⊥,⊥),
   (3, (10,3,0,10),
   (4, (10,3,0,10),
   (5, (10,3,0,10),
   (3, (10,3,1,7),
   (4, (10,3,1,7),
   (5, (10,3,1,7),
   (3, (10,3,2,4),
   ...
   (3, (10,3,3,1),
   (6, (10,3,3,1)  〉
```

¿Cómo son las ejecuciones del diagrama "reducido" a partir de estos mismos estados?

2.9 Semántica operacional de un programa GCL

Dado S, un programa GCL, se pueden considerar dos predicados Q,R sobre el espacio de estados del programa.

Se dice que S es *totalmente correcto con respecto a (la precondición)* Q *y (a la poscondición)* R

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

si para toda ejecución de S a partir de un estado que satisfaga Q, ésta termina normalmente en un estado que satisface R (la notación {Q} S {R} es un *tripla de Hoare*).

Como un caso particular muy importante, se dice que S termina (desde Q), si

$$\{Q\}\ S\ \{true\}.$$

Ejemplos

Considérese el programa S, del ejemplo de 2.8. Se puede comprobar que valen:

a {Q: x1=10 ∧ x2=3} S {R: y1=3 ∧ y2=1}

b {Q: x2≠0} S {R: true}

c {Q: x1=A ∧ x2=B ∧ x2≠0} S {R: y1 = A **div** B ∧ y2 = A **mod** B}.

3 SEMÁNTICA AXIOMÁTICA

La semántica axiomática de GCL corresponde a una teoría que tiene como modelo la semántica operacional descrita en **2**.

La conexión entre los dos conceptos está en la noción de *corrección* de un programa (o instrucción) con respecto a una especificación. Las ideas básicas se encuentran en [Gri1993] 1.6, 10 y 12.6. En [Car1993] se describe esta teoría en mayor detalle, a lo largo del capítulo 2.

La idea de usar una semántica axiomática es evitar el cálculo operacional de la corrección. En cambio, la teoría axiomática provee axiomas y reglas de inferencia para mostrar que una tripla de Hoare se cumple. Para saber qué tipo de axiomas y reglas de inferencia deben darse, basta con definir éstos para las instrucciones básicas del lenguaje (**`skip`**, **`abort`**, asignación) y establecer cómo las instrucciones complejas apoyan su semántica en la de sus partes.

En [Gri1993] se sugiere el concepto de *precondición más débil* (inglés: *weakest precondition*, wp) para definir la corrección de las instrucciones de asignación. La wp es una función que asigna un predicado a una pareja ⟨programa, condición⟩ , i.e.,

$$\begin{array}{rccc} \texttt{wp:} & \texttt{Programas} \times \texttt{Predicados} & \rightarrow & \texttt{Predicados} \\ & \langle \texttt{S,R} \rangle & \mapsto & \texttt{wp(S|R)} \end{array}$$

de modo que wp(S|R) es un predicado tal que:

(i) {wp(S|R)} S {R}

(ii) {Q} S {R} ≡ (Q ⇒ wp(S|R))

En el modelo operacional se puede mostrar que un tal predicado existe siempre y es único. De esta forma, se puede definir una wp para cada instrucción de GCL, lo que definirá la corrección de cualquier programa.

En la práctica se muestran teoremas que prueban la corrección de instrucciones en condiciones bastante generales; estos teoremas se pueden usar como reglas de inferencia para calcular la correción de programas. El mecanismo deductivo resultante es el llamado *Cálculo de Hoare*. Son axiomas y reglas de inferencia que, agregadas a la lógica de predicados, sirven para mostrar la corrección de programas.

3.1 Semántica axiomática para GCL

Partiendo de la noción de wp ya referida, el cálculo de corrección para GCL se establece definiendo wp(S|R) para cada posibles instrucción S y poscondición R. Esta es una definición necesariamente recursiva, dado que las instrucciones de GCL son, precisamente, aquellos constructos sintácticos que satisfacen la gramática del lenguaje. De hecho, lo que se hace es definir wp(S|R) donde S varía sobre las diferentes categorías sintácticas de instrucciones y R es cualquier condición.

Además de la semántica propia de las instrucciones, hay ciertas condiciones generales que debe satisfacer una función wp y que reflejan alguna propiedad que tiene sentido exigir, basándose en el modelo operacional. Estas propiedades generales no dependen, en rigor, de GCL.

3.1.1 Propiedades generales para wp

Ax: Ley del milagro excluido

wp(S| false) ≡ false

El axioma se interpreta entendiendo que, para cualquier instrucción S, no hay manera de que una ejecución termine normalmente en un estado que cumpla el predicado `false`.

Ax : Ley de $\wedge$-distributividad

$wp(S\ |(\forall i|\ Q.i : R.i)) \equiv (\forall i|\ Q.i : wp(S|R.i)).$

El axioma se justifica de manera natural al interpretar la semántica operacional de las partes.

Teo: $\wedge$-distributividad finita

$wp(S|R_1 \wedge R_2) \equiv wp(S|R_1) \wedge wp(S|R_2)$

Esta es una versión finita del axioma de $\wedge$-distributividad.

Teo: Ley de monotonía

$R_1 \Rightarrow R_2 \ \vdash\ wp(S|R_1) \Rightarrow wp(S|R_2)$

Es una consecuencia de la $\wedge$-distributividad finita.

Teo: Ley de $\vee$-semidistributividad

$wp(S\ |(\exists i|\ Q.i : R.i)) \Leftarrow (\exists i|\ Q.i : wp(S|R.i)).$

Es una consecuencia de la ley de monotonía. Una versión finita:

$wp(S|\ R1 \vee R2) \Leftarrow\ wp(S|R1) \vee wp(S|R2).$

Teo: Continuidad[2]

Sea $(PRE, \sqsubseteq)$ el conjunto de predicados con el orden parcial,

$$Q \sqsubseteq R \ \equiv\ R \Rightarrow Q, \text{ para } Q, R \in PRE.$$

Considérese una cadena ascendente de predicados $(Q_1, Q_2, \ldots)$. Entonces se cumple que:

$$wp(S|(sup\ i|:\ Q_i)) \equiv (sup\ i|:\ wp(S|Q_i)).$$

3.1.1 wp para instrucciones GCL

Ax: `skip`

$wp(\mathbf{skip}|R) \equiv R$

Ax: Asignación

$wp(x := e|\ R) \equiv R[x := e]$

---

[2] Se dice que una función f sobre un conjunto ordenado $(X, |)$ es *continua* si, para cada cadena ascendente $(x_1, x_2, \ldots)$ se tiene que $f.(sup\ i|:\ x_i) = (sup\ i|:\ f.x_i)$.

En rigor, se puede extender esta definición para considerar posibles indefiniciones al calcular e. Si dom.e es un predicado que es true en aquellos estados en los que e está bien definido, el axioma debe ser:

$wp(x:= e|\ R) \equiv dom.e \wedge R[x:= e]$

Por otro lado, la definición incluye asignaciones a elementos de arreglos, si se considera, además, la notación

$(b;\ i_1:e_1,\ldots,i_m:e_m)$

para un arreglo igual al arreglo b, pero que en las posiciones $i_1,\ldots,i_r$ (todas ellas diferentes) tiene los valores $e_1,\ldots,e_r$. Entonces, se define

$wp(b[i]:= e|\ R) \equiv R[b:= (b;i:e)]$

Finalmente, las asignaciones paralelas o simultáneas se entienden definidas por el axioma:

$wp(x_1,\ldots,x_m:= e_1,\ldots,e_m\ |R) \equiv R[x_1,\ldots,x_m:= e_1,\ldots,e_m].$

Ax: Condicional

Para la instrucción condicional IF definida como

IF: **if** $B_1 \rightarrow S_1$
[] $B_2 \rightarrow S_2$
…
[] $B_n \rightarrow S_n$
fi

se define

$wp(IF|\ R) \equiv BB \wedge (\forall i|\ 1\leq i\leq n\ :\ B_i \Rightarrow wp(S_i|R)).$

Ax: Iteraciones

Para la instrucción iterativa DO definida como

DO: **do** $B_1 \rightarrow S_1$
[] $B_2 \rightarrow S_2$
…
[] $B_n \rightarrow S_n$
od

se define

$wp(DO|\ R) \equiv (\exists i|\ 1\leq i\ :\ wp(IF_i|R)).$

donde:

$IF_0 \cong$ **if** $\neg BB \rightarrow$ **skip fi**
$IF_{k+1} \cong$ **if** $\neg BB \rightarrow$ **skip** , para $k\geq 0$.
[] $BB \rightarrow IF;\ IF_k$
fi

La definición de wp de DO no es constructiva. Esto, en general, dificulta la implementación de algoritmos que soporten la verificación de programas con base en wp's. En cambio se pueden establecer teoremas que garanticen la corrección de programas con ciclos con respecto a una especificación dada si, además, se añade suficiente información (invariante, argumentos de terminación) a la documentación de los ciclos.

3.2 Cálculo de Hoare

En el cálculo de Hoare hay axiomas y reglas de inferencia que deben valer independientemente del lenguaje de programación[3]. Otras reglas, específicas al lenguaje que se maneje, se pueden establecer en cada caso, una vez se conocen las correspondientes a un lenguaje algorítmico como GCL.

Propiedades generales

[RI F]

$$\frac{\{Q\}\ S\ \{false\}}{Q \equiv false}$$

[RI pre]

$$\frac{Q \Rightarrow Q1,\ \{Q1\}\ S\ \{R\}}{\{Q\}\ S\ \{R\}}$$

[RI pos]

$$\frac{\{Q\}\ S\ \{R1\},\ \ R1 \Rightarrow R}{\{Q\}\ S\ \{R\}}$$

[Ax Distr/∀**]** $\quad (\forall i |\ i \in I : \{Q\}\ S\ \{Ri\}) \ \equiv\ \ \{Q\}\ S\ \{(\forall i |\ i \in I : Ri)\}$

Propiedades del lenguaje GCL

[Ax skip] $\quad Q \Rightarrow R \ \equiv\ \{Q\}$ **skip** $\{R\}$

[Ax abort] $\quad \neg Q \ \equiv\ \{Q\}$ **abort** $\{R\}$

[RI ;]

$$\frac{\{Q\}\ S1\ \{R1\},\ \{R1\}\ S2\ \{R\}}{\{Q\}\ S1;S2\ \{R\}}$$

[Ax ≔] $\quad Q \Rightarrow R[x \coloneqq e] \ \equiv\ \{Q\}\ x \coloneqq e\ \{R\}$

[RI IF]

$$\frac{Q \Rightarrow BB,\ (\forall i |\ 1 \le i \le n : \{Q \wedge Bi\}\ Si\ \{R\})}{\{Q\}\ IF\ \{R\}}$$

[RI DO]

$$\frac{\{Q\}\ INIC\ \{P\},\ P \wedge \neg BB \Rightarrow R,\ (\forall i |\ 1 \le i \le n : \{P \wedge Bi\}\ Si\ \{P\}),\ \text{Terminación}}{\{Q\}\ INIC;\ \{inv\ P\}\ DO\ \{R\}}$$

La regla **[RI DO]** es el teorema del DO del Capítulo 12 de [Gri1993]. Una manera coloquial de recordarla es:

---

[3] Se habla de lenguajes *imperativos*, donde hay un concepto de *estado* que puede ser modificado –usualmente- mediante instrucciones de asignación.

$$\frac{\texttt{P vale antes, P sirve, P invariante, Terminación}}{\texttt{\{Q\} INIC; \{inv P\} DO \{R\}}}$$

La condición de terminación también se estudia en [Gri1993]. De hecho, se dan condiciones suficientes para mostrar terminación usando funciones cota definidas sobre enteros o, en general, sobre órdenes bien fundados.

BIBLIOGRAFÍA

[Gri1983] Gries, D. *The Science of Programming,* Springer Verlag, 1983.

[Car1993] Cardoso, R., *Verificación y desarrollo de programas*, Ediciones Ecoé-Uniandes, 1993.

[Gri1993] Gries, D., *A logical approach to discrete Math*, Springer Verlag, 1993.