UNIVERSITAT DE BARCELONA
Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

VERIFICACIÓN DE PROGRAMAS

Autora: Ana Rodríguez Martínez

Director: Dr. Juan Carlos Martínez Alonso
Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 17 de enero de 2024

Abstract

Hoare's logic is one of the main methods for verifying the partial correctness of programs. With it, it can be reasoned whether, starting from some initial conditions and assuming the execution ends, an output that meets the expected final conditions can be obtained. The main objective of this paper is to study the soundness, incompleteness and completeness in the sense of Cook of Hoare's system. In addition, other methods related to program verification and program termination will be presented, accompanied by practical examples.

Resumen

La lógica de Hoare es uno de los métodos principales para la verificación de la corrección parcial de programas. Con ella se puede razonar si partiendo de unas condiciones iniciales y suponiendo que termina la ejecución, se puede obtener una salida que cumpla las condiciones finales esperadas. El objetivo principal de este trabajo es estudiar la corrección, la incompletitud y la completitud en el sentido de Cook del sistema de Hoare. Además se presentarán otros métodos relacionados con la verificación y la terminación de programas acompañados de ejemplos prácticos.

2024 Mathematics Subject Classification. 68Q60, 68N30, 03B10

Esta memoria se ha desarrollado con LaTeXutilizando el editor TexMaker.

Todos los programas se han escrito en lenguaje C conforme al estándar ISO/IEC 9899:2018, también conocido como C18.

Agradecimientos

En primer lugar, quiero agradecer a mi tutor, Juan Carlos Martínez Alonso, por su gran ayuda a lo largo de la elaboración de este trabajo y por haberme propuesto este tema, el cual he acabado disfrutando tanto.

También, quiero agradecer a mi familia por el apoyo constante que me han dado. Y en especial a mi madre, que ha hecho todo lo que ha podido y más por ayudarme.

Finalmente, quiero darles las gracias a mis amigas de *Computabilitat i Complexitat*, a mis amigos de Ludo UB y a Víctor por haber hecho de estos últimos años una experiencia tan maravillosa.

Muchísimas gracias.

Índice

1. Introducción

En principio, las consecuencias de ejecutar un programa en cualquier entorno se pueden encontrar partiendo del propio código del programa a partir del racionamiento. Por ello, es de gran interés desarrollar y estudiar métodos que permitan razonar sobre cómo será la salida de los programas y si coincide con los resultados que se esperan obtener de ellos.

En 1967, Robert Floyd propuso una técnica para verificar la corrección de la salida de los programas, conocida como el método de aserción inductiva o método de Floyd, que indica que "un diagrama de flujo es correcto respecto a las aserciones finales e iniciales si se pueden encontrar aserciones intermedias válidas" [8]. Para demostrar que es correcto, pues, basta con encontrar las condiciones intermedias y demostrar la corrección de los fragmentos del programa entre las condiciones intermedias [2].

En 1969, basándose parcialmente en el método de Floyd, Charles Antony Richard Hoare [9] introdujo un nuevo método para la verificación de programas: la lógica de Hoare. Un sistema formal que proporciona una serie de reglas de verificación para razonar sobre la corrección de programas imperativos, el cual ha tenido un gran impacto en los métodos utilizados para el diseño y verificación de programas [1].

A pesar de que el método de Hoare supone una mejora respecto al método de Floyd, sigue sin garantizar la terminación del programa. Por lo que, en caso de querer analizar un programa a partir de estos métodos, es necesario complementar la verificación de la corrección del programa con métodos para la verificación de la terminación de este. Como son por ejemplo, el método de los contadores o el método de Floyd. No obstante, como el problema de parada es irresoluble, ni estos métodos, ni cualquier otro, sirven de forma general para estudiar la parada de todos los programas.

Cabe remarcar que, aunque el estudio de la verificación abarca toda clase de programas, este trabajo se centra en la verificación de programas iterativos.

Estructura de la Memoria

En esta memoria se tratan los conceptos que se han considerado más relevantes para ofrecer una introducción a la verificación de programas.

En la sección 2 se presenta el análisis de programas. Se explica y ejemplifica el método de Floyd para analizar la corrección de programas y se dan nociones para el cálculo de la complejidad de los programas. También se presenta el problema de la parada y se demuestra que no puede existir una función que determine la parada de todo programa.

En la sección 3 se introducen los lenguajes de primer orden. Se dan los conceptos de sintaxis y semántica que se necesitarán en la sección 4.

La sección 4 es el cuerpo del trabajo. En ella se definen las reglas del cálculo de Hoare y se aplican a la verificación de diversos programas. Y, además, se demuestra la corrección, la incompletitud y la completitud en el sentido de Cook del sistema de Hoare.

Y en la sección 5 se tratan dos métodos para estudiar la terminación de los programas: el método de Floyd y el método de los contadores. Para poder explicar el método de Floyd, primero se introducen ciertos conceptos de conjuntos ordenados.

2. Análisis de programas

Un programa consiste en una secuencia finita de instrucciones u órdenes no ambiguas basadas en un lenguaje de programación que una computadora interpreta para resolver un problema o tarea específica.
Las características esenciales que un programa debe poseer son corrección, eficiencia y claridad. Para analizar un programa se deben estudiar las características de este. En este primer apartado, se ofrece una introducción al análisis de la corrección, la eficiencia temporal y la parada del programa.

2.1. Corrección de programas

La corrección es una de las características esenciales que debe poseer un programa y consiste en garantizar que el programa desempeñe la labor para la que fue desarrollado. A simple vista, podría parecer que el testeo de los programas es una herramienta suficiente para garantizar la corrección del programa. Sin embargo, no es así, pues solo puede demostrar que este contiene errores. Para demostrar que es correcto, se deberían testear todas las entradas admisibles, pero no es factible para la gran mayoría de programas. En su lugar se pueden utilizar determinados métodos de verificación.

A continuación se presentará el método de Floyd (véase [2]) -también conocido como método de aserción inductiva- como método para garantizar la corrección de los programas iterativos. Y más adelante se mostrará el método de Hoare de demostración formal de la corrección de programas.

Previamente, debe conocerse el concepto de invariante.

Definición 2.1. Un **invariante** es un predicado que describe los distintos estados por los que pasa un bucle en un programa, estableciendo una relación entre las variables que intervienen en este. Debe cumplir las siguientes condiciones:

- Se satisface antes de empezar el bucle, es decir, antes de la primera iteración.
- Se mantiene al ejecutar el cuerpo del bucle.
- Se cumple al salir del bucle.

Para aplicar **el método de Floyd** se debe dibujar el diagrama de flujo del programa e identificar la aserción inicial y final. Si el programa tiene bucles, se definen los invariantes de estos. Para demostrar la corrección del programa, se hace una inducción sobre el número de vueltas de los bucles y se demuestra sus invariantes.

Ejemplo 2.2. Verificación del algoritmo de la división entera a través del método de Floyd. Ver figura 1.

El invariante del bucle es $(x = qy + r \wedge r \geqslant 0)$. En la vuelta 0 , partiendo de la aserción inicial $(x \geqslant 0 \wedge y > 0)$ y haciendo las asignaciones $q := 0$; $r := x$, se cumple el invariante.

Supóngase que se cumple el invariante en la vuelta k, partiendo de (1) y haciendo la comprobación $r \geqslant y$ para entrar en el bucle. Si no cumple la condición, se obtiene $(x = qy + r \wedge y > r \geqslant 0)$, que coincide con la post condición. En el caso de que se cumpla y entre en el punto (2), se tiene $(x = qy + r \wedge r \geqslant 0 \wedge r \geqslant y)$. Tras el cuerpo del bucle, $r := r - y$ y $q := q + 1$, se puede expresar r y q en función de sus nuevos valores como $r + y$ y $q - 1$ respectivamente. Por lo que, al volver a (1) se tiene que $(x = (q-1)y + (r+y) \wedge (r+y) \geqslant 0 \wedge (r+y) \geqslant y)$, que es equivalente a $(x = qy - y + r + y \wedge r + y \geqslant 0 \wedge r + y \geqslant y)$, se observa que el invariante $(x = qy + r \wedge r \geqslant 0)$ se sigue cumpliendo, llegando a la vuelta k+1.

Como se observa que para cualquier entrada el algoritmo parará y llegará a la postcondición $(x = qy + r \wedge 0 \leqslant r \leqslant y)$, queda demostrada la corrección.

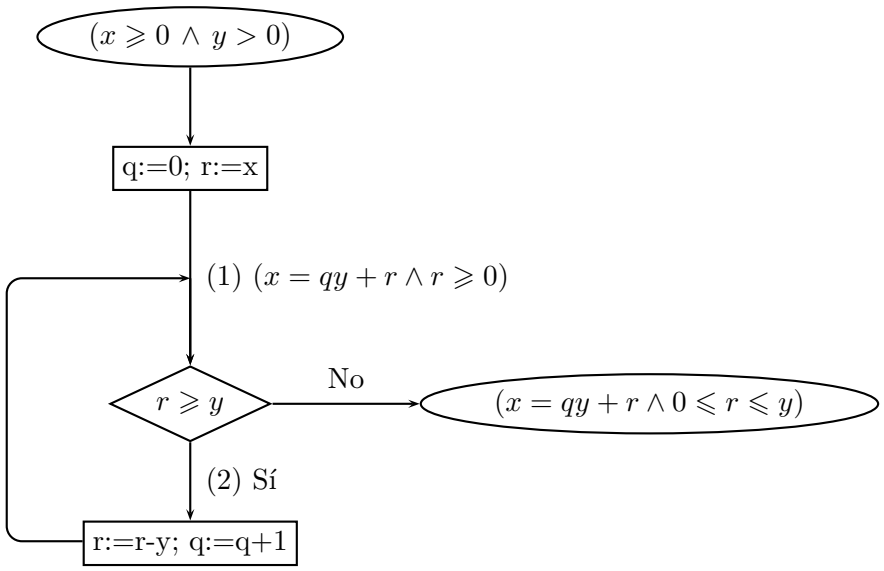

Figura 1: Diagrama de flujo de un programa para calcular divisiones enteras.

Ejemplo 2.3. Verificación de un algoritmo que calcula el máximo común divisor, a través del método de Floyd. Ver figura 2

Al principio, en la vuelta 0, es evidente que partiendo de la aserción inicial $(x \geqslant 0 \wedge y \geqslant 0)$ y haciendo las asignaciones $a := x$; $b := y$, se satisface el invariante del bucle $(mcd(a, b) = mcd(x, y))$.

Supóngase que se cumple el invariante en la vuelta k y partiendo de (1). Si al hacer la comprobación $a \neq b$ se sale del bucle, se consigue que $a = b$, por lo tanto $mcd(a, b) = a = b$, llegando a la postcondición $(mcd(x, y) = a = b)$. En el caso de que $a \neq b$ se cumpla y entre en el punto (2), se llega a $(mcd(a, b) = mcd(x, y)) \wedge (a \neq b)$. Se presentan dos casos según si $a > b$.

Si se cumple $a > b$ (3), se tiene $(mcd(a, b) = mcd(x, y)) \wedge (a > b)$ y tras $a := a - b$, se puede expresar a en función de su nuevo valor $a + b$. Al volver a (1) con $(mcd(a + b, b) =$

$mcd(x,y)) \wedge (a+b > b)$, ya que $mcd(a+b,b) = mcd(a,b)$, se obtiene $mcd(x,y) = mcd(a,b)$, por lo que se llega a la vuelta k+1.

Si no se cumple $a > b$ (4) , entonces $(mcd(a,b) = mcd(x,y)) \wedge (a < b)$ y tras $b := b - a$, se puede expresar b en función de su nuevo valor $b + a$. Al volver a (1) con $(mcd(a, b+a) = mcd(x,y)) \wedge (a < b+a)$ se obtiene $mcd(x,y) = mcd(a,b)$, por lo que se llega a la vuelta k+1.

Como se observa que para cualquier entrada se saldrá del bucle, se llegara a la post-condición $(mcd(x,y) = a = b)$.

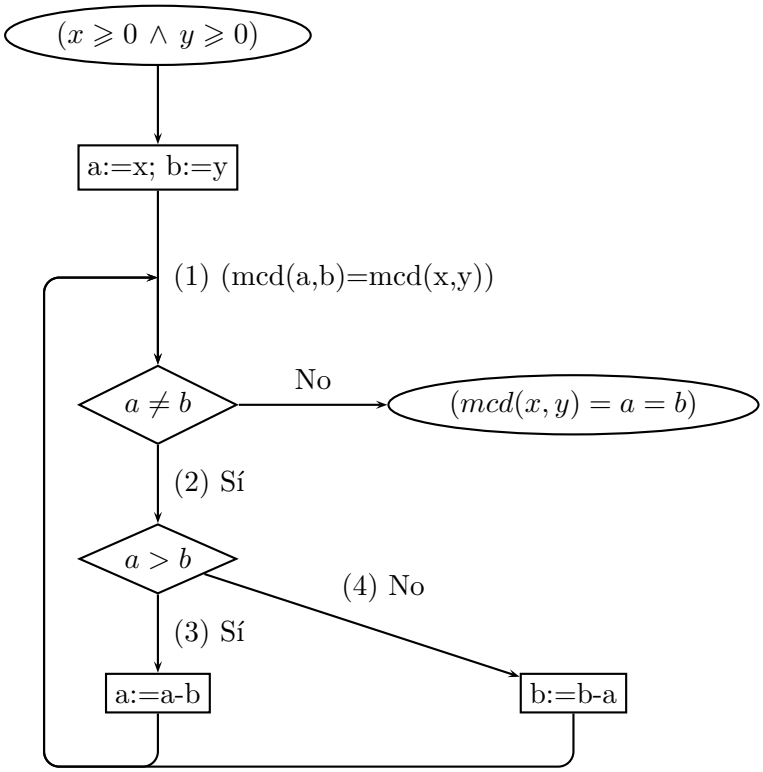

Figura 2: Diagrama de flujo de un programa para calcular el máximo común divisor.

Ejemplo 2.4. Verificación de un algoritmo con dos bucles anidados, que calcula la potencia x^y, a través del método de Floyd. Ver figura 3

A diferencia de los ejemplos anteriores, este programa tiene dos bucles anidados, por lo que se deberán realizar dos inducciones sobre el número de iteraciones en cada bucle.

Se iniciará resolviendo el bucle interno. Si se cumple el invariante del primer bucle $(z * u^v = x^y \wedge v \geqslant 0)$ y se entra en este cumpliendo $v \neq 0$, entonces en la vuelta 0 del bucle (2) se cumple $(z * u^v = x^y \wedge v > 0)$.

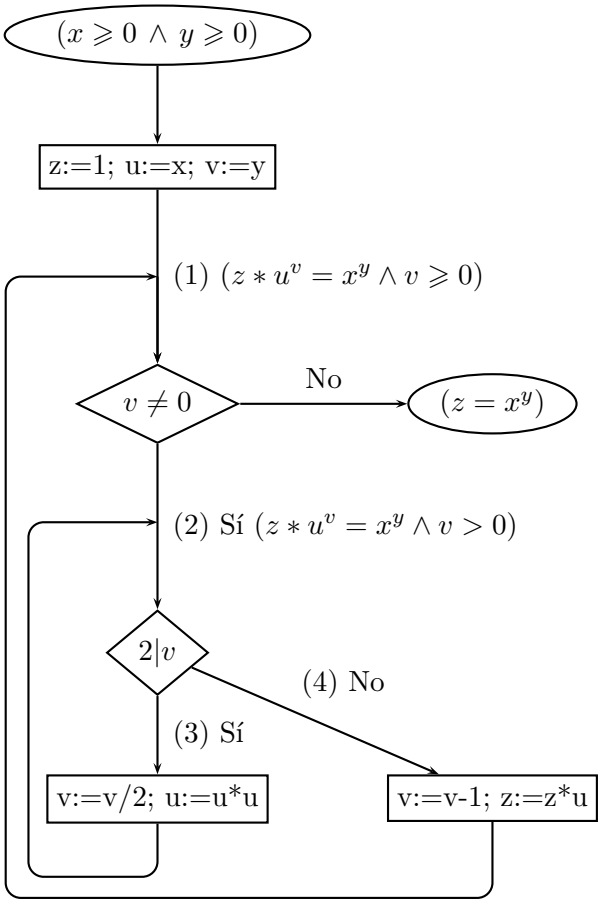

Figura 3: Diagrama de flujo de un programa para el calcular potencias.

Supóngase que se cumple el invariante en la vuelta k del bucle interno y partiendo del invariante $(z * u^v = x^y \wedge v > 0)$. Si al hacer la comprobación $2|v$, no se cumple, pues v es impar, entonces se sale del bucle llegando a (4) con $(z * u^v = x^y \wedge v > 0) \wedge (2 \not| v)$. Si se cumple $2|v$, se llega a (3) con $(z * u^v = x^y \wedge v > 0) \wedge (2|v)$. Tras hacer las asignaciones v:=v/2 y u:=u*u, se puede expresar v en función de su nuevo valor como 2*v, pues v es par, y expresar u como $u^{\frac{1}{2}}$. Por todo ello, $(z * (u^{\frac{1}{2}})^{2v} = x^y \wedge 2v > 0)$, que equivale al invariante de (2) $(z * u^v = x^y \wedge v > 0)$, llegando a la vuelta k+1 del bucle (2).

Ahora resuélvase el bucle externo. En la vuelta 0, con las condiciones iniciales $x, y \geqslant 0$ y haciendo las asignaciones z:=1; u:=x; v:=y, se observa que se satisface el invariante $(z * u^v = x^y \wedge v \geqslant 0)$.

Supóngase que se cumple el invariante en la vuelta k del bucle externo. Si no se cumple $v \neq 0$, entonces $(z * u^v = x^y \wedge v = 0)$. Por lo tanto, como $x^y = z * u^v = z * u^0 = z * 1 = z$, se llega a la conclusión final $(z = x^y)$. Si sí se cumple $v \neq 0$, se alcanza al invariante del segundo bucle $(z * u^v = x^y \wedge v > 0)$. Cuando se salga del segundo bucle en (4), se obtendrá $(z * u^v = x^y \wedge v > 0) \wedge (2 \not| v)$ y las asignaciones v:=v-1 y z:=z*u. Expresando v y z en función de sus nuevos valores v+1 y z/u, se obtiene $(\frac{z}{u} * u^{v+1} = x^y \wedge v + 1 > 0)$.

Como $x^y = \frac{z}{u} * u^{v+1} = \frac{z}{u} * u * u^v = z * u^v$ y al estar trabajando con enteros $v + 1 > 0$ es equivalente a $v \geqslant 0$, se llega al invariante del bucle (1) ($z * u^v = x^y \wedge v \geqslant 0$) y a la vuelta k+1.

2.2. Análisis de complejidad

El análisis de la complejidad de un programa consiste en estudiar cómo de eficientemente consume sus recursos. Por ejemplo, el tiempo de ejecución o la memoria necesaria para ejecutarse. En este apartado se explicarán algunos de los conceptos que aparecen en [10] y [5].

En vez de centrarse en el coste real de un algoritmo, implementado y ejecutado en un lenguaje y una máquina en particular, se le da relevancia a cómo depende el consumo de recursos con respecto al tamaño de los datos de la entrada que el programa tiene que procesar.

Denotando, pues, $T(n)$ a **la función del tiempo de ejecución** de un programa con entrada de tamaño n, $T(n)$ representa el número de operaciones elementales que realiza el algoritmo para obtener la solución. Se llamará **funciones de coste** a la funciones de la forma $f : \mathbb{N} \to \mathbb{R}^*$, donde $\mathbb{R}^* = \{0\} \cup \mathbb{R}^+$. Toman como argumento números naturales, representando el tamaño de los datos, y devuelven números reales no negativos que representan el coste correspondiente. f y g representarán funciones de coste.

Como el tiempo de ejecución exacto depende de varios factores, cuando se dice que $T(n)$ es una determinada función f(n), se quiere indicar que $T(n)$ es proporcional a f(n). Pues, de estas funciones no interesan los valores concretos que pueda generar, sino su forma de crecimiento, lo rápido que crecen cuando se aumenta el tamaño de los datos. Por ello, se utiliza la notación asimptótica.

Definición 2.5. El orden de la función de coste f, escrito $O(f)$, es la clase formada por todas aquellas funciones de coste que estén acotadas superiormente por un múltiplo de f. Es decir, $g \in O(f)$ si existe un número natural n_0 y un número real positivo c tal que para todo $n \geqslant n_0$, se tiene que $g(n) \leqslant cf(n)$.

Cuando se dice que $T(n) \in O(f(n))$, la velocidad de crecimiento de $T(n)$ está acotada superiormente por $f(n)$.

Cálculo de la complejidad

Cuando se analiza la complejidad de un programa, es necesario poder expresar su función de tiempo de ejecución y obtener el orden de compejidad. A la hora de descomponer un programa en sus componentes, se pueden considerar cuatro tipos diferentes: operaciones elementales, secuencias de operaciones, sentencias condicionales y bucles.

Operaciones elementales Es el conjunto de operaciones básicas de un lenguaje imperativo como C. Como es, por ejemplo, la asignación para datos de tipo primitivo, operaciones aritméticas básicas, operaciones boleanas básicas, acceso a la posición de un

vector o fichero, las instrucciones de lectura, y escritura y la instrucción return. Cada una de estas operaciones tiene un coste constante, por lo que pertenecen a $O(1)$.

Secuencias de operaciones El tiempo total de $I_1; I_2$ es la suma de los tiempos de I_1 y I_2.

Sentencias condicionales de la forma if (B) I_1 else I_2; o if (B) I_1;. El coste es la suma del coste de evaluar la condición booleana B más el tiempo de la instrucción I_1 o I_2 con la complejidad más alta.

Bucles de la forma while (B) I;. Para calcular el coste del bucle, hay que calcular el número de iteraciones del bucle y multiplicarlo por el tiempo de ejecución de la iteración de mayor coste más el coste de evaluar la condición del bucle.

Para analizar la complejidad de un algoritmo recursivo es frecuente que aparezcan funciones de coste recursivas, llamadas recurrencias, donde se separa el coste básico del coste del caso recursivo. En este último, se usa la misma función para representar el coste de la llamada recursiva.

Ejemplos 2.6. Cálculo de la complejidad de dos programas diseñados para calcular el n-ésimo término de la sucesión de Fibonacci.

Versión iterativa:

```
int fibon1(int n){
        int i, x, y, z;
        x=1;
        y=1;
        for (i=3; i<= n; i++){
                z=y;
                y = x+y;
                x=z;
        }
        return y;
}
```

Se calcula $T_1(n)$, la primera parte de la función fibon1 es una serie de operaciones básicas que tendrá un coste constante c_1. El coste del bucle será el coste del cuerpo y de la evaluación de condición, que es una constante c_2, y será multiplicado por el número de iteraciones, en este caso se puede simplificar y decir que serán n iteraciones. Por lo que el coste del bucle será $c_2 n$. Finalmente, se le suma el coste del return, el cual es una constante c_3.
El tiempo de ejecución $T_1(n)$ del programa fibon1 tiene complejidad lineal, pues:

$$T_1(n) = c_1 + nc_2 + c_3 \in O(n)$$

Versión recursiva:

```
int fibon2(int n){
        if (n == 1 || n == 2){
```

```
            return 1;
        }else{
            return fibon2(n-1) + fibon2(n-2);
        }
}
```

Si se denota $T_2(n)$ al tiempo de ejecución del programa fibon2, se tiene que:

$$T_2(n) = \begin{cases} c_1 & si \quad n \leqslant 2 \\ \\ T_2(n-1) + T_2(n-2) + c_2 & si \quad n \geqslant 3 \end{cases}$$

Por lo tanto:

$$\begin{aligned} T_2(n) &= T_2(n-1) + T_2(n-2) + c_2 = T_2(n-2) + T_2(n-3) + c_2 + T_2(n-2) + c_2 \\ &= 2T_2(n-2) + T_2(n-3) + 2c_2 = 2(T_2(n-3) + T_2(n-4) + c_2) + T_2(n-3) + 2c_2 \\ &= 3T_2(n-3) + 2T_2(n-4) + 4c_2 = 3(T_2(n-4) + T_2(n-5) + c_2) + 2T_2(n-4) + 4c_2 \\ &= 5T_2(n-4) + 3T_2(n-5) + 7c_2 = 5(T_2(n-5) + T_2(n-6) + c_2) + 3T_2(n-5) + 7c_2 \\ &= 8T_2(n-5) + 5T_2(n-6) + 12c_2 = ... = \\ &\text{fibonacci}(i+1)T_2(n-i) + \text{fibonacci}(i)T_2(n-(i+1)) + (\text{fibonacci}(i+2) - 1)c_2 \end{aligned}$$

donde fibonacci(i) es el i-ésimo término de la sucesión de Fibonacci. Tomando i=n-2 se llega al caso básico

$$\begin{aligned} T_2(n) &= \text{fibonacci}(n-1)T_2(2) + \text{fibonacci}(n-2)T_2(1) + (\text{fibonacci}(n) - 1)c_2 \\ &= \text{fibonacci}(n-1)c_1 + \text{fibonacci}(n-2)c_1 + (\text{fibonacci}(n) - 1)c_2 \\ &= (\text{fibonacci}(n-1) + \text{fibonacci}(n-2))c_1 + (\text{fibonacci}(n) - 1)c_2 \\ &= \text{fibonacci}(n)c_1 + (\text{fibonacci}(n) - 1)c_2 \end{aligned}$$

Por la fórmula de Binet, se sabe que

$$\text{fibonacci}(n) = \frac{a^n - (1-a)^n}{\sqrt{5}} \text{ donde } a = \frac{1+\sqrt{5}}{2}$$

Por lo tanto:

$$T_2(n) = \frac{a^n - (1-a)^n}{\sqrt{5}}(c_1 + c_2) - c_2$$

Como $1 < a < 2$, se deduce que $T_2(n) \in O(2^n)$

2.3. El problema de parada

Además de estudiar la corrección y la complejidad, una característica importante a analizar de un programa es su capacidad de parada. La parada de los programas sin bucles está garantizada, pero al introducir operaciones iterativas cabe la posibilidad de que el programa, bajo ciertas entradas, jamás finalice. Se dice que un programa para, si el cómputo de P termina en un número finito de pasos.

Sería interesante ver si hay alguna función general para todos los programas. Hay muchas formas de demostrar que esta función no existe; en este caso se probará codificando un programa en C mediante una tira de 0's y 1's, basándose en la demostración utilizada en [7].

Codificación a 0's y 1's de un programa en C

Se define la función inyectiva:

$$\text{Código: } \{\text{programas en C}\} \to \{0,1\}^* \tag{2.1}$$

En C se distinguen los siguientes símbolos: letras, dígitos, símbolos de puntuación y palabras reservadas. A continuación, se les asigna sus correspondientes codificaciones 2.1.

Codificación de las letras:

Colocando las minúsculas en su orden clásico, seguidas de las mayúsculas según la misma ordenación, se puede numerar cada letra con un número que comienza por 10 y es proseguido de tantos 1's como indique su posición comenzando a por el puesto 1.

a	b	...	z	A	...	Z
101	1011	...	$101...^{(25)}1$	$101...^{(21)}1$	...	$101...^{(52)}1$

Codificación de los dígitos:

Al igual que con las letras, se ordenan los dígitos crecientemente y se les asigna un número, que en este caso empezará por 100 para diferenciarlos de las letras y será proseguido por tantos 1's como indique el dígito más uno extra.

0	1	2	...	9
1001	10011	100111	...	$1001...^{(8)}1$

Codificación de los símbolos de puntuación:

Considerando los símbolos de puntuación que pueden aparecer en un programa escrito en C -como son . ; , ( ) !-, se les puede asignar un orden arbitrario y un número que empiece por 1000 y continúe con tantos 1's como indique su posición.

σ_1	σ_2	σ_3	...	σ_n
10001	100011	1000111	...	$10001...^{(n-2)}1$

Codificación de las palabras reservadas:

El lenguaje C tiene palabras reservadas que se deben diferenciar de una simple tira de caracteres como son las palabras "while", "for", double", "void"… Se les puede asignar un orden y un número que empiece por 10000 y sea seguido por tantos 1's como indique su posición.

σ_1	σ_2	σ_3	...	σ_m
100001	1000011	10000111	...	$100001...^{(m-2)}1$

Observación 2.7. Considerando un programa P en C, como una tira de símbolos $P = \sigma_1...\sigma_k$, se define la codificación de P como:

$$\text{Código}(P) = \text{Código}(\sigma_1)...\text{Código}(\sigma_k) \tag{2.2}$$

Problema de la parada

¿Existe un programa que decida, dado un programa cualquiera P y una entrada x, si P para sobre la entrada x?

A continuación, se demostrará que un programa así no puede existir.

Demostración:

Se define la función HALT: $\mathbb{N}^2 \to \{0,1\}$ como una función que, dado un programa codificado y una entrada, devuelve 1 si el programa para sobre la entrada y 0 en el caso contrario.

$$HALT(x,y) = \begin{cases} 1 \quad si \quad \text{y =Código(P) y el programa P para sobre la entrada x} \\ \\ 0 \qquad\qquad\qquad\qquad\qquad\qquad \text{en el caso contrario} \end{cases}$$

Supóngase que el problema de parada es resoluble. Entonces, se tiene que la función HALT es computable.

Por lo tanto, se puede definir la función H(x) := HALT(x,x), que también será computable.

Considérese la siguiente función F:

```
int F (int x1){

        int x2;
        x2 = H(x1);
        if(x2 == 0){
                return x2;
        }else{
                while(x2 == 1){}
        }
}
```

La función F o bien escribe 0 o entra en un bucle infinito. Sea y_0 = Código(F) y x un natural, entonces:

$$\text{HALT}(x, y_0) = 1 \leftrightarrow F(x) = 0$$

Por lo tanto,

$$\text{HALT}(y_0, y_0) = 1 \leftrightarrow F(y_0) = 0$$

Por la definición de F se sabe que

$$F(y_0) = 0 \leftrightarrow H(y_0) = 0$$

Por lo que se obtiene

$$\text{HALT}(y_0, y_0) = 1 \leftrightarrow F(y_0) = 0 \leftrightarrow H(y_0) = 0 \leftrightarrow \text{HALT}(y_0, y_0) = 0$$

llegando a contradicción. Por lo tanto, la función HALT no es computable.

3. Lenguajes de primer orden

Los conceptos y teoremas de esta sección pueden consultarse en [7], serán esenciales para comprender más adelante la lógica de Hoare.

La lógica de primer orden o lógica de predicado se puede entender como una extensión de la lógica proposicional, incluyendo conceptos adicionales como los cuantificadores, los símbolos de función y los símbolos de predicados, y permitiendo, así, formalizar aserciones que no se pueden expresar con la lógica proposicional.

Definición 3.1. El **alfabeto** de un lenguaje de primer orden $\mathcal{A}_S$ contiene los siguientes símbolos:

- Variables: $v_0, v_2, v_3, ..., v_n, ...$
- Constantes: a, b, c, ..., a_0, b_0, c_0, ..., a_n, b_n, c_n, ...
- Símbolos de función: f, g, h, f_0, g_0, h_0, ..., f_n, g_n, h_n, ...
- Símbolos de relación o predicados: A, B, ..., Z, A_0, B_0, ..., Z_0, ..., A_n, B_n, ..., Z_n, ...
- Conectores $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
- Cuantificadores $\exists, \forall$
- Símbolos auxiliares: paréntesis y comas

Definición 3.2. Un **vocabulario** es un conjunto S de símbolos de constantes, símbolos de función y símbolos de relación. Por tanto, S es la unión disyunta de C_S, F_S y R_S, donde C_S es el conjunto de constantes de S, F_S el conjunto de símbolos de función de S y R_S el conjunto de símbolos de relación de S.

Tanto los símbolos de función como los símbolos de relación tienen asociado un número natural $n \geqslant 1$, su ariedad. Para denotar que un símbolo de función f es n-ario se escribirá f^n y para denotar que un símbolo de relación R es n-ario se escribirá R^n. Se denota como VAR al conjunto de variables. Se denota por $\mathcal{A}$ el conjunto de símbolos de la variables, conectores, cuantificadores y auxiliares. $\mathcal{A}_S = \mathcal{A} \cup S$ donde S es un vocabulario.

3.1. Términos y fórmulas

Definición 3.3. Los **S-términos** o términos de $\mathcal{A}_S$ son una sucesión finita de símbolos de $C_S \cup F_S \cup VAR$ que se pueden definir por recursión de la siguiente forma:

Las variables $x_i \in VAR$ son términos.

Las constantes $c \in C_S$ son términos.

Si f es un símbolo de función n–ario $f \in F_S$ y $t_1, ..., t_n$ son términos, entonces $ft_1...t_n$ es un término.

Definición 3.4. Una **ecuación** es una expresión de la forma $t_1 \doteq t_2$ donde t_1 y t_2 son términos de S.

Definición 3.5. Las **fórmulas atómicas** de $\mathcal{A}_S$ son las ecuaciones y las expresiones de la forma $Rt_1...t_n$ donde R es un símbolo de relación n-ario de S y t_1 ... t_n son términos de A_S.

Definición 3.6. Las **S-fórmulas** o fórmulas son las expresiones que se definen recursivamente de la siguiente manera:

Las fórmulas atómicas son fórmulas

Si φ es una fórmula, entonces $\neg\varphi$ también es una fórmula.

Si φ, ψ son fórmulas y $* \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$, entonces $(\varphi * \psi)$ es una fórmula.

Si φ es una fórmula y x una variable, entonces $\exists x\varphi$ y $\forall x\varphi$ son fórmulas.

Definición 3.7. Se define L_S, o L(S), el **lenguaje de primer orden** asociado al alfabeto $\mathcal{A}_S$ como el conjunto de fórmulas de $\mathcal{A}_S$.

Definición 3.8. Una **ocurrencia** de una variable x en la fórmula φ es **libre** si no está dentro del alcance de ningún cuantificador que ligue la variable x. Si lo está, se trata de una ocurrencia **ligada**.

Definición 3.9. Las **variables libres** de una fórmula φ son las que tienen al menos una ocurrencia libre en φ. Las **variables ligadas** de φ son las que tienen todas las ocurrencias ligadas en φ.

Cuando $x_1, ..., x_n$ es una lista de variables distintas, se usa la notación t$(x_1, ..., x_n)$para referirse a un término cuyas variables aparecen en esa lista y $\varphi(x_1, ..., x_n)$ para referirse a una fórmula cuyas variables libres aparecen en la lista. Las fórmulas sin variables libres se llaman enunciados o **sentencias**. También se llaman fórmulas abiertas a las fórmulas con variables libres y fórmulas cerradas a los enunciados.

3.2. Sustituciones

Se busca definir la sustitución de forma que φ exprese lo mismo de x que ψ de t, siendo ψ la fórmula obtenida al sustituir en φ, la x por t.

Ejemplo 3.10. En este ejemplo se ilustra el objetivo y se muestra por qué hay que proceder con cuidado

$$\varphi := \exists \text{z } \text{z+z} \equiv x$$

En $\mathbb{N}$, la fórmula dice que x es par. Si se sustituye x por y en φ, se obtiene la fórmula $\exists z$ z+z $\equiv y$, que dice que y es par. Pero si se sustituye x por z, se obtiene la fórmula $\exists z$ z+z $\equiv z$, la cual no está diciendo que z sea par. El significado ha sido alterado porque donde la ocurrencia de la variable x era libre, la ocurrencia de z está ligada.

Definición 3.11. $t\binom{t_0...t_r}{x_0...x_r}$ es el término obtenido al sustituir simultáneamente cada aparición de las variables $x_0...x_r$ por los términos $t_0...t_r$ respectivamente.

Definición 3.12. $\varphi\binom{t_0...t_r}{x_0...x_r}$ es la fórmula obtenida al sustituir simultáneamente cada aparición libre de las variables $x_0...x_r$ por los términos $t_0...t_r$. Se define la sustitución con la siguiente inducción:

$$[t'_0 \equiv t'_1]\binom{t_0...t_r}{x_0...x_r} := t'_0\binom{t_0...t_r}{x_0...x_r} \equiv t'_1\binom{t_0...t_r}{x_0...x_r}$$

$$[Rt'_0...t'_{n-1}]\binom{t_0...t_r}{x_0...x_r} := Rt'_0\binom{t_0...t_r}{x_0...x_r}...t'n-1\binom{t_0...t_r}{x_0...x_r}$$

$$[\neg\varphi]\binom{t_0...t_r}{x_0...x_r} := \neg[\varphi\binom{t_0...t_r}{x_0...x_r}]$$

$$[\varphi * \psi]\binom{t_0...t_r}{x_0...x_r} := \varphi\binom{t_0...t_r}{x_0...x_r} * \psi\binom{t_0...t_r}{x_0...x_r}$$

donde $* \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$.

Supóngase que $x_{i_0}, ..., x_{i_{s-1}}$ son todas las variables x_i entre $x_0, ..., x_r$ que son libres en $\exists x\varphi$ y $x_i \neq t_i$. Entonces, se define:

$$[\exists x\varphi]\binom{t_0...t_r}{x_0...x_r} := \exists u[\varphi\binom{t_{i_0}, ..., t_{i_{s-1}}u}{x_{i_0}...x_{i_{s-1}}x}]$$

donde u es la variable x si no hay ocurrencia de x en $t_{i_0}, ..., t_{i_{s-1}}$ y si no, es la primera variable la la lista $v_0, v_1, v_2...$, la cual no tiene ocurrencias en $\varphi, t_{i_0}, ..., t_{i_{s-1}}$.

Al introducir la variable u, se garantiza que la ocurrencia de ninguna variable en $t_{i_0}, ..., t_{i_{s-1}}$ caiga en el alcance de un cuantificador. En el caso de que no haya ningún x_i tal que sea libre en $\exists x\varphi$ y $x_i \neq t_i$, se tiene que s=0 y $[\exists x\varphi]\binom{t_0...t_r}{x_0...x_r} := \exists x[\varphi\binom{x}{x}]$

Ejemplo 3.13. Sean f y P binarios.

$[Pv_0fv_1v_2]\binom{v_2,v_0,v_1}{v_1,v_2,v_3} = Pv_0fv_2v_0$

$[\exists v_0Pv_0fv_1v_2]\binom{v_4,fv_1v_1}{v_0,v_2} = \exists v_0[Pv_0fv_1v_2\binom{fv_1v_1,v_0}{v_2,v_0}]$, pues v_0 no tiene ocurrencia en fv_1v_1, $\exists v_0[Pv_0fv_1v_2\binom{fv_1v_1,v_0}{v_2,v_0}] = \exists v_0Pv_0fv_1fv_1v_1$

$[\exists v_0Pv_0fv_1v_2]\binom{v_0,v_2,v_3}{v_1,v_2,v_0} = \exists v_4[Pv_0fv_1v_2\binom{v_0,v_4}{v_1,v_0}]$, pues $t_{i_0}, ..., t_{i_{s-1}}$ es v_0. $\exists v_4[Pv_0fv_1v_2\binom{v_0,v_4}{v_1,v_0}] = \exists v_4Pv_4fv_0v_2$

Hasta el momento, se ha centrado la atención en la sintaxis. A continuación, se procederá a explicar la semántica de primer orden.

3.3. Estructuras e interpretaciones

Definición 3.14. Una estructura de vocabulario S o **S-estructura** es un par $\mathfrak{U} = (A, \mathfrak{a})$, donde A es un conjunto no vacío llamado el universo de la estructura y $\mathfrak{a}$ es una aplicación cuyo dominio es S y cumple las siguientes condiciones:

Para cada constante c en S, $\mathfrak{a}(c)$ es un elemento de A.

Para cada símbolo de función n-ario f en S, $\mathfrak{a}(f)$ es una función n-aria de A $\mathfrak{a}(f) : A^n \rightarrow A$.

Para cada símbolo de relación n-ario R en S, $\mathfrak{a}(R)$ es una relación n-aria de A $\mathfrak{a}(R) \subseteq A^n$.

Notación 3.15. Si $\mathfrak{U} = (A, \mathfrak{a})$, se suele usar la notación $\xi^{\mathfrak{U}}$ o ξ^A en vez de $\mathfrak{a}(\xi)$ para cualquier símbolo $\xi \in S$. Por tanto, se puede poner $\mathfrak{U} = (A, \xi^{\mathfrak{U}})_{\xi \in S}$.

Definición 3.16. Una **asignación** en $\mathfrak{U}$ es una función α cuyo dominio es el conjunto de variables de $\mathcal{A}$ y cuyo recorrido es un subconjunto de A.
$\alpha : \{v_n \mid n \in \mathbb{N}\} \to A$

Definición 3.17. Una **S-interpretación** o interpretación del lenguaje L=L(S) o L-interpretación es un par $\mathfrak{I} = (\mathfrak{U}, \alpha)$, donde $\mathfrak{U}$ es una S-estructura y α es una asignación en $\mathfrak{U}$.

Definición 3.18. Si $\mathfrak{I}$ es una S-interpretación con una asignación α en $\mathfrak{U}$ y x es una variable, se define $\mathfrak{I}_a^x$, como la interpretación $\mathfrak{I}_a^x = (\mathfrak{U}, \alpha_a^x)$, donde α_a^x es una asignación igual a α, salvo en la imagen de la variable x, que pasa a ser $\alpha_a^x(x) = a$

$$\alpha_a^x(y) = \begin{cases} \alpha(y) & si \quad y \neq x \\ \\ a & si \quad y = x \end{cases}$$

La satisfactibilidad caracteriza la relación que hay cuando una fórmula resulta ser verdadera bajo una interpretación.

Definición 3.19. Como paso preliminar, para un término t y una interpretación $\mathfrak{I} = (\mathfrak{U}, \alpha)$, se define la evaluación de t bajo $\mathfrak{I}$ como un elemento $\mathfrak{I}(t)$ del dominio A de la siguiente forma:

Para una variable x de $\mathfrak{U}$, $\mathfrak{I}(x) := \alpha(x)$

Para una constante c de L, $\mathfrak{I}(c) := c^{\mathfrak{U}}$

Para un símbolo de función n-ario $f \in L$ y los términos $t_0, ..., t_{n-1}$, $\mathfrak{I}(f(t_0...t_{n-1})) := f^{\mathfrak{U}}(\mathfrak{I}(t_0), ..., \mathfrak{I}(t_{n-1}))$.

Definición 3.20. Usando la inducción sobre la fórmula φ, se da una definición a la relación $\mathfrak{I}$ es un **modelo** de φ, escrita como $\mathfrak{I} \models \varphi$:

Sea $\mathfrak{I} = (\mathfrak{U}, \alpha)$ una interpretación cualquiera.

$\mathfrak{I} \models t_0 \equiv t_1$ si y solo si $\mathfrak{I}(t_0) = \mathfrak{I}(t_1)$

$\mathfrak{I} \models Rt_0...t_{n-1}$ si y solo si $R^{\mathfrak{U}}\mathfrak{I}(t_0)...\mathfrak{I}(t_{n-1})$

$\mathfrak{I} \models \neg\varphi$ si y solo si $\mathfrak{I}$ no modeliza φ (i.e. $\mathfrak{I} \not\models \varphi$)

$\mathfrak{I} \models (\varphi \wedge \psi)$ si y solo si $\mathfrak{I} \models \varphi$ y $\mathfrak{I} \models \psi$

$\mathfrak{I} \models (\varphi \vee \psi)$ si y solo si $\mathfrak{I} \models \varphi$ o $\mathfrak{I} \models \psi$

$\mathfrak{I} \models (\varphi \to \psi)$ si y solo si, si $\mathfrak{I} \models \varphi$, entonces $\mathfrak{I} \models \psi$

$\mathfrak{I} \models (\varphi \leftrightarrow \psi)$ si y solo si, $\mathfrak{I} \models \varphi$ si y solo si $\mathfrak{I} \models \psi$

$\mathfrak{I} \models \forall x\varphi$ si y solo si para todo $a \in A$ $\mathfrak{I}_a^x \models \varphi$

$\mathfrak{I} \models \exists x\varphi$ si y solo si hay algún $a \in A$ tal que $\mathfrak{I}_a^x \models \varphi$

$\mathfrak{I} \models \varphi$ también se puede leer como $\mathfrak{I}$ modeliza φ o $\mathfrak{I}$ satisface φ

Observación 3.21. Si $\mathfrak{I} \models \varphi$, entonces φ es un enunciado verdadero bajo la interpretación $\mathfrak{I}$.

Notación 3.22. Para una S-fórmula φ y una S-interpretación $\mathfrak{I} = (\mathfrak{U}, \alpha)$, la validez de φ bajo $\mathfrak{I}$ tan solo depende de la asignación de las variables libres de φ y la interpretación de los símbolos de S en $\mathfrak{U}$.

Se puede utilizar la notación $\mathfrak{U} \models \varphi(\alpha(v_1), ..., \alpha(v_n))$ en vez de $(\mathfrak{U}, \alpha) \models \varphi$. Si φ es una sentencia, se escribe $\mathfrak{U} \models \varphi$.

Definición 3.23. Sea Φ un conjunto de fórmulas. Se dice que $\mathfrak{I}$ es un **modelo** de Φ, escrito $\mathfrak{I} \models \Phi$, si $\mathfrak{I} \models \varphi$ para todo $\varphi \in \Phi$.

Definición 3.24. Una fórmula φ es **satisfactible**, escrito Sat φ, si existe una interpretación $\mathfrak{I}$ tal que $\mathfrak{I} \models \varphi$.

Un conjunto de fórmulas Φ es **satisfactible**, escrito Sat Φ, si existe una interpretación $\mathfrak{I}$ tal que $\mathfrak{I} \models \Phi$.

Si Φ no es satisfactible, se dice que Φ es **insatisfactible**.

3.4. La relación de consecuencia lógica

Definición 3.25. Sean $\varphi \in L$ y $\Phi \subseteq L$. Se dice que φ es **consecuencia lógica** de Φ, escrito $\Phi \models \varphi$, si toda interpretación $\mathfrak{I}$, que es un modelo de Φ, es también un modelo de φ.

Definición 3.26. Los **axiomas** del cálculo de primer orden en un vocabulario L son los siguientes:

- $(\varphi \to (\psi \to \varphi))$
- $((\varphi \to (\psi \to \chi)) \to ((\varphi \to \psi) \to (\varphi \to \chi))$
- $((\neg\varphi \to \neg\psi) \to ((\neg\varphi \to \psi) \to \varphi))$
- $(\forall x\varphi \to \varphi(\binom{t}{x}))$, donde x es libremente sustituible por t en φ
- $(\forall x(\varphi \to \psi) \to (\forall x\varphi \to \forall x\psi))$
- $(\varphi \to \forall x\varphi)$, donde x no está libre en φ
- $x \doteq x$
- $(\varphi(\binom{t}{x}) \to (t \doteq t' \to \varphi(\binom{t'}{x})))$, donde φ es una fórmula atómica de L, x es una variable y t,t' son términos de L

Definición 3.27. Modus Ponens es la única regla de inferencia del cálculo que permite inferir ψ a partir de φ y $(\varphi \to \psi)$.

Definición 3.28. Una **deducción** con premisas en Φ es una sucesión finita de fórmulas $\varphi_1 ... \varphi_n$, donde cada φ_i puede ser o un axioma o una fórmula de Φ o una fórmula obtenida mediante Modus Ponens de fórmulas anteriores.

Los teoremas y definiciones que aparecerán a continuación serán muy importantes para entender el apartado de la lógica de Hoare. Serán especialmente relevantes en los subapartados de Corrección y Completitud de H. Para un desarrollo más detallado, consultar [7].

Definición 3.29. Una fórmula φ es **deducible** de Φ y se escribe $\Phi \vdash \varphi$, si existe una deducción con premisas en Φ cuya última fórmula es φ.

Teorema 3.30. ***Teorema de Corrección y Completitud***

Sea φ una fórmula y Φ un conjunto de fórmulas. Entonces $\Phi \models \varphi$ si y solo si $\Phi \vdash \varphi$.

Definición 3.31. Un conjunto de fórmulas es **finitamente satisfactible** si todo subconjunto finito suyo es satisfactible.

A partir del Teorema de Corrección y Completitud, se obtiene en forma de corolario el Teorema de Compacidad.

Teorema 3.32. ***Teorema de Compacidad*** *Si un conjunto de fórmulas es finitamente satisfactible, entonces es satisfactible.*

Definición 3.33. Sea $\mathfrak{U}$ una S-estructura. Se define la teoría de la estructura $\mathfrak{U}$ como $Th(\mathfrak{U}) := \{\varphi : \varphi \text{ es una S-sentencia } \wedge \mathfrak{U} \models \varphi\}$.

4. Lógica de Hoare

El método de Hoare, o lógica de Hoare, es un sistema formal que proporciona una serie de reglas para razonar sobre la corrección de programas iterativos. Este método, parcialmente basado en el método de Floyd [9], presenta varias ventajas ante su predecesor, como no necesitar representar el programa como un organigrama, ser más fácil de aplicar para programas con bucles anidados y ser sistematizable. Gracias a que es sitematizable, algunos lenguajes de programación, como Dafny, CSP, o JML, incorporan la lógica de Hoare para estudiar la corrección de los programas.

4.1. Programas iterativos

Sea L=L(S) un lenguaje de primer orden. Se denota por $\varphi, \psi, \chi, ...$ a las fórmulas de L; la letra b denotará una S-fórmula libre de cuantificadores; y el conjunto de variables y términos se representará como V y T respectivamente. Sea $\mathscr{P}(S)$ la clase de programas definida por las siguientes reglas:

1. Si $x \in V$ y $t \in T$, entonces $(x := t) \in \mathscr{P}$
2. Si $P_1, P_2 \in \mathscr{P}$, entonces $(P_1; P_2) \in \mathscr{P}$
3. Si $b \in L$ es libre de cuantificadores y $P \in \mathscr{P}$, entonces (if (b) P)$\in \mathscr{P}$
4. Si $b \in L$ es libre de cuantificadores y $P_1, P_2 \in \mathscr{P}$, entonces (if (b) P_1 else P_2) $\in \mathscr{P}$
5. Si $b \in L$ es libre de cuantificadores y $P \in \mathscr{P}$, entonces (while (b) P)$\in \mathscr{P}$.

A los elementos de $\mathscr{P}$ se les llama programas iterativos [1].

4.2. El cálculo de Hoare

Tal y como se explica en [1], en la lógica de Hoare se manejan ternas del tipo $\{\varphi\}P\{\psi\}$, llamados **programas asertados**, donde se fija un vocabulario S y el lenguaje L=L(S), $P \in \mathscr{P}(S)$ y $\varphi, \psi \in L(S)$ describen el estado inicial y final del programa P. El sistema lógico donde φ y ψ se expresan se llama lenguaje de aserciones.

Intuitivamente, el programa asertado, $\{\varphi\}P\{\psi\}$, es verdad si y solo si, cuando la precondición φ se verifica antes de la ejecución de P y P termina, entonces la postcondición ψ se verifica después de la ejecución de P.

El cálculo de Hoare consta de las siguientes reglas [10]:

(R0) Regla del programa vacío:

$$\frac{}{\{\varphi\}\ \{\varphi\}}$$

donde φ es una S-fórmula.

(R1) Regla de la asignación:

$$\frac{}{\{\varphi\binom{t}{x}\}x := t\{\varphi\}}$$

donde φ es una S-fórmula, x aparece libre en φ y t es un término. Las apariciones libres de x en φ se sustituyen por t. Después de ejecutarse $x := t$, x satisface la misma propiedad que satisfacía t antes de ocurrir la asignación.

Ejemplo 4.1.

$$\{x + y > 2\}x := x + y\{x > 2\}$$
$$\{\exists j(x = 2j)\}z := x\{\exists j(z = 2j)\}$$

(R2) Regla de la composición:

$$\frac{\{\varphi\}P_1\{\chi\}\ ,\ \{\chi\}P_2\{\psi\}}{\{\varphi\}(P_1; P_2)\{\psi\}}$$

donde φ, χ, ψ son S-fórmulas y $P_1, P_2 \in \mathscr{P}(S)$. La composición secuencial de dos instrucciones P_1 y P_2, denotada $P_1; P_2$, se utiliza para encadenar cómputos. El estado final producido por P_1 se convierte en el estado inicial de P_2. Esta regla puede ser generalizada del siguiente modo:

$$\frac{\{\varphi\}P_1\{\chi_1\}, \{\chi_1\}P_2\{\chi_2\}, ..., \{\chi_{n-1}\}P_n\{\psi\}}{\{\varphi\}(P_1; P_2; ...; P_n)\{\psi\}}$$

Ejemplo 4.2. Aplicando la regla (R2), se tiene, por ejemplo, $\{x = a \wedge y = l\}$ aux:=x; x:=y; y:=aux $\{y = a \wedge x = l\}$.

(R3) Regla de la consecuencia:

$$\frac{(\varphi \to \varphi_1)\ ,\ \{\varphi_1\}P\{\psi_1\}\ ,\ (\psi_1 \to \psi)}{\{\varphi\}P\{\psi\}}$$

donde $\varphi, \varphi_1, \psi, \psi_1$ son S-fórmulas y $P \in \mathscr{P}(S)$.

(R4) Regla condicional simple:

$$\frac{\{\varphi \wedge b\}P\{\psi\}\ ,\ ((\varphi \wedge \neg b) \to \psi)}{\{\varphi\}\ \text{(if (b) P)}\ \{\psi\}}$$

donde b es una S-fórmula libre de cuantificadores; φ, ψ son S-fórmulas y $P \in \mathscr{P}(S)$.

Ejemplo 4.3. Teniendo $\{a = 3e \wedge e + 1 \pmod 2 = 0\}$ e:=e+1 $\{a = 3e \wedge e \pmod 2 = 0\}$ como $(a = 3e \wedge e \pmod 2 = 0) \to a \pmod 2 = 0$, aplicando (R3), se obtiene $\{a = 3e \wedge e + 1 \pmod 2 = 0\}$ e:=e+1 $\{a \pmod 2 = 0\}$. Como $(a = 3e \wedge \neg(e + 1 \pmod 2 = 0)) \to a \pmod 2 = 0$, entonces, aplicando (R4), se obtiene $\{a = 3e\}$ (if (e+1 (mod 2)=0) e:=e+1) $\{a \pmod 2 = 0\}$.

(R5) Regla condicional compuesta:

$$\frac{\{\varphi \wedge b\}P_1\{\psi\}\ ,\ \{\varphi \wedge \neg b\}P_2\{\psi\}}{\{\varphi\}\ \text{(if (b) } P_1 \text{ else } P_2\ )\ \{\psi\}}$$

donde b es una S-fórmula libre de cuantificadores; φ, ψ son S-fórmulas; y $P_1, P_2 \in \mathscr{P}(S)$.

Ejemplo 4.4. Teniendo $\{x - y \geqslant 0 \wedge x - y = x - y\}$ z:=x-y $\{x - y \geqslant 0 \wedge z = x - y\}$, como $x \geqslant y \rightarrow (x - y \geqslant 0 \wedge x - y = x - y)$, y ya que $(x - y \geqslant 0 \wedge z = x - y) \rightarrow z = |x - y|$, aplicando (R3), se obtiene $\{x \geqslant y\}$ z:=x-y $\{z = |x - y|\}$. Análogamente, se obtiene $\{\neg x \geqslant y\}$ z:=y-x $\{z = |x - y|\}$. Aplicando (R5), se obtiene { true } if $(x \geqslant y)$ z:=x-y; else z=y-x; $\{z = |x - y|\}$.

(R6) Regla iterativa while:

$$\frac{\{\varphi \wedge b\}P\{\varphi\}}{\{\varphi\} \text{ (while (b) P) } \{\varphi \wedge \neg b\}}$$

donde b es una S-fórmula libre de cuantificadores, la S-fórmula φ es el invariante del bucle y $P \in \mathscr{P}(S)$.

Ejemplo 4.5. Teniendo $\{0 \leqslant x + 1 \leqslant n \wedge y + y = 2^{x+1} \wedge x \neq n\}$ x:=x+1; y:=y+y $\{0 \leqslant x \leqslant n \wedge y = 2^x\}$, como $y = 2^x \rightarrow y + y = 2^{x+1}$ y también $(0 \leqslant x \leqslant n \wedge x \neq n) \rightarrow 0 \leqslant x + 1 \leqslant n$, al aplicar (R3), se tiene $\{0 \leqslant x \leqslant n \wedge y = 2^x \wedge x \neq n\}$ x:=x+1; y:=y+y $\{0 \leqslant x \leqslant n \wedge y = 2^x\}$, que cumple las hipótesis iniciales de (R6), siendo $0 \leqslant x \leqslant n \wedge y = 2^x$ el invariante y $x \neq n$ la expresión booleana. Al aplicar (R6), se obtiene $\{0 \leqslant x \leqslant n \wedge y = 2^x\}$ while $(x \neq n)$ x:=x+1; y:=y+y $\{0 \leqslant x \leqslant n \wedge y = 2^x \wedge x = n\}$.

A partir de las reglas anteriores, se obtienen las reglas derivadas de Hoare, como son la regla del *do while* y la regla del *switch case*.

Ejemplo 4.6. Derivación de la **regla iterativa do while**.

$$\frac{\{\varphi\}P\{\psi\},\ (\psi \wedge b) \rightarrow \varphi}{\{\varphi\} \text{ do P while (b) } \{\psi \wedge \neg b\}}$$

donde b es una S-fórmula libre de cuantificadores; φ y ψ son S-fórmulas; $P \in \mathscr{P}(S)$ y (do P while (b)) $\equiv$ (P; while (b) P).
Gracias a las premisas, se tiene

$$(1)\ \{\varphi\}P\{\psi\}$$

y

$$(2)\ (\psi \wedge b) \rightarrow \varphi$$

Aplicando la regla de la consecuencia (R3) a (1) y (2), se obtiene

$$(3)\ \{\psi \wedge b\}P\{\psi\}$$

Al aplicar la regla iterativa (R6) a (3), se tiene

$$(4)\ \{\psi\} \text{ while (b) P } \{\psi \wedge \neg b\}$$

Finalmente, al utilizar la regla de la composición (R2) a (1) y (4), se obtiene

$$(5)\ \{\varphi\} \text{ do P while (b) } \{\psi \wedge \neg b\}$$

Ejemplo 4.7. Derivación de la **regla condicional switch case**.

$$\frac{\{\varphi \wedge b_1\}P_1\{\psi\}, ..., \{\varphi \wedge \neg b_1 \wedge \neg b_2 \wedge ... \wedge b_{n-1}\}P_{n-1}\{\psi\}, \{\varphi \wedge \neg b_1 \wedge \neg b_2 \wedge ... \wedge \neg b_{n-1}\}P_n\{\psi\}}{\{\varphi\} \text{ switch (x) case } c_1\text{: } P_1 \text{ break; ... case } c_{n-1}\text{: } P_{n-1} \text{ break; default: } P_n \ \{\psi\}}$$

donde φ y ψ son S-fórmula, b_i es una S-fórmula libre de cuantificadores de la forma $x \doteq c_i$, siendo c_i una constante, $P_i \in \mathscr{P}(S)$ y la instrucción (switch (x) case c_1: P_1 break; ... case c_{n-1}: P_{n-1} break; default: P_n) es equivalente a (if (b_1) P_1 else (... (if (b_{n-1}) P_{n-1} else P_n)...).

Aplicando la regla de la condicional compuesta (R5) a $\{\varphi \wedge \neg b_1 \wedge \neg b_2 \wedge ... \wedge b_{n-1}\}P_{n-1}\{\psi\}$ y $\{\varphi \wedge \neg b_1 \wedge \neg b_2 \wedge ... \wedge \neg b_{n-1}\}P_n\{\psi\}$, se obtiene $\{\varphi \wedge \neg b_1 \wedge \neg b_2 \wedge ... \wedge b_{n-2}\}$ if (b_{n-1}) P_{n-1} else P_n $\{\psi\}$.

Al resultado obtenido se le aplica (R5) junto la premisa que hace referencia al programa P_{n-2}. Este proceso se repite hasta alcanzar la premisa $\{\varphi \wedge b_1\}P_1\{\psi\}$, obteniendo finalmente

$$\{\varphi\} \text{ (if } (b_1) \ P_1 \text{ else (... (if } (b_{n-1}) \ P_{n-1} \text{ else } P_n \text{)...) } \{\psi\}$$

Como (if (b_1) P_1 else (... (if (b_{n-1}) P_{n-1} else P_n)...) es equivalente a (switch (x) case c_1: P_1 break; ... case c_{n-1}: P_{n-1} break; default: P_n), se tiene

$$\{\varphi\} \text{ (switch (x) case } c_1\text{: } P_1 \text{ break; ... case } c_{n-1}\text{: } P_{n-1} \text{ break; default: } P_n) \ \{\psi\}$$

Ejemplo 4.8. Sea maxComDiv un programa escrito en C que calcula el máximo común divisor de dos naturales distintos de cero.

```
void maxComDiv(){
        int a, b, x, y;
        scanf("%d",&x);
        scanf("%d",&y);
        a=x;
        b=y;
        while(a!=b){
                if(a>b){
                        a=a-b;
                }else{
                        b=b-a;
                }
        }
        printf("el mcd es %d",a);
        return;
}
```

Para verificar este programa, se demostrará $\{0 < x \wedge 0 < y\}$ *maxComDiv* $\{a = b = mcd(a, b) = mcd(x, y)\}$ a través del método de Hoare de la siguiente forma:

A través de las reglas (R1) y (R2), se tiene

(1) $\{0 < x \wedge 0 < y \wedge x = x \wedge y = y\}$ a:=x; b:=y $\{0 < x \wedge 0 < y \wedge a = x \wedge b = y\}$

Se identifica mcd(x,y)=mcd(a,b) como el invariante del bucle. Como $(0 < x \wedge 0 < y) \rightarrow (0 < x \ \wedge \ 0 < y \ \wedge \ x = x \ \wedge \ y = y)$ y $(0 < x \ \wedge \ 0 < y \ \wedge \ a = x \ \wedge \ b = y) \rightarrow (mcd(x, y) = mcd(a, b))$, aplicando (R3), se obtiene

$$(2)\ \{0 < x \wedge 0 < y\}\ \text{a:=x; b:=y}\ \{mcd(x, y) = mcd(a, b)\}$$

A través de (R1), se tiene

$$(3)\ \{mcd(x, y) = mcd(a - b, b)\}\ \text{a:=a-b}\ \{mcd(x, y) = mcd(a, b)\}$$

y

$$(4)\ \{mcd(x, y) = mcd(a, b - a)\}\ \text{b:=b-a}\ \{mcd(x, y) = mcd(a, b)\}$$

Utilizando la definición de máximo común divisor, se tiene que si a>b, entonces $mcd(a, b) = mcd(a - b, b)$, pues mcd(a,b) es un divisor común de a-b y b, y mcd(a-b,b) es un divisor común de a y b. Por lo tanto, se cumple la implicación $(mcd(x, y) = mcd(a, b) \wedge a \neq b \wedge a > b) \rightarrow (mcd(x, y) = mcd(a - b, b))$. Análogamente, se demuestra que $(mcd(x, y) = mcd(a, b) \wedge a \neq b \wedge a \leqslant b) \rightarrow (mcd(x, y) = mcd(a, b - a))$.

Aplicando (R3) a (3) y (4), usando estas implicaciones, se tiene

$$(5)\ \{mcd(x, y) = mcd(a, b) \wedge a \neq b \wedge a > b\}\ \text{a:=a-b}\ \{mcd(x, y) = mcd(a, b)\}$$

y

$$(6)\ \{mcd(x, y) = mcd(a, b) \wedge a \neq b \wedge a \leqslant b\}\ \text{b:=b-a}\ \{mcd(x, y) = mcd(a, b)\}$$

Aplicando (R5) a (5) y (6), se tiene

$$(7)\ \{mcd(x, y) = mcd(a, b) \wedge a \neq b\}\ \text{if (a>b) a:=a-b else b:=b-a}\ \{mcd(x, y) = mcd(a, b)\}$$

Al aplicar (R6) a (7), se tiene

$$(8)\ \{mcd(x, y) = mcd(a, b)\}\ (\text{while } (a \neq b) \text{ if (a>b) a:=a-b else b:=b-a }) \ \{mcd(x, y) = mcd(a, b) \wedge a = b\}$$

Como $(mcd(x, y) = mcd(a, b) \wedge a = b) \rightarrow (a = b = mcd(x, y) = mcd(a, b))$, aplicando (R3) a (8) y (R2) a (2) y (8), finalmente se tiene

$$(9)\ \{mcd(x, y) = mcd(a, b)\}\ maxComDiv\ \{a = b = mcd(x, y) = mcd(a, b)\}$$

Ejemplo 4.9. Considérese potencia un programa escrito en C que calcula, x^y siendo x e y naturales.

```
void potencia(){
        int x, y, z, u, v;
        scanf("%d",&x);
        scanf("%d",&y);
        z=1;
        u=x;
        v=y;
        while(v!=0){
```

```
            while(v%2==0){
                /* division entera v//2 */
                v=v/2;
                u=u*u;
            }
            v=v-1;
            z=z*u;
        }
        printf("la potencia de x elevado a y es %d",z);
        return;
}
```

Para verificar este programa, se demostrará $\{0 < x, y\}$ *potencia* $\{z = x^y\}$ a través del método de Hoare de la siguiente forma:
Se identifica $zu^v = x^y$ como el invariante del bucle externo. A través de las reglas (R1) y (R2), se tiene

$$(1)\ \{0 < x, y \wedge 1x^y = x^y\}\ \text{z:=1; u:=x; v:=y}\ \{zu^v = x^y\}$$

El invariante del bucle interno es $zu^v = x^y \wedge v \neq 0$. Aplicando (R1) y (R2), se tiene

$$(2)\ \{0 \neq v \wedge z(u^2)^{\frac{v}{2}} = x^y\}\ \text{v:=v/2; u:=}u^2\ \{0 \neq v \wedge zu^v = x^y\}$$

Como $(0 \neq v \wedge zu^v = x^y \wedge 2 \mid v) \rightarrow (0 \neq v \wedge z(u^2)^{\frac{v}{2}} = x^y)$, se aplica (R3) a (2), obteniendo $\{0 \neq v \wedge zu^v = x^y \wedge 2 \mid v\}$ v:=v/2; u:=u^2 $\{0 \neq v \wedge zu^v = x^y\}$. Al aplicar (R6) a esta condición, se tiene

$$(3)\ \{0 \neq v \wedge zu^v = x^y\}\ \text{(while } (2 \mid v)\ \text{v:=v/2; u:=uu)}\ \{0 \neq v \wedge zu^v = x^y \wedge 2 \nmid v\}$$

Aplicando las reglas (R1) y (R2), se tiene

$$(4)\ \{zuu^{v-1} = x^y\}\ \text{v:=v-1; z:=zu}\ \{zu^v = x^y\}$$

Como $(0 \neq v \wedge 2 \nmid v \wedge zu^v = x^y) \rightarrow (zuu^{v-1} = x^y)$, al aplicar (R3) a (3), se obtiene $\{0 \neq v \wedge zu^v = x^y\}$ (while $(2 \mid v)$ v:=v/2; u:=uu) $\{zuu^{v-1} = x^y\}$. Al aplicar (R2) a este programa asertado y a (4), se tiene

$$(5)\ \{zu^v = x^y \wedge v \neq 0\}\ \text{(while } (2 \mid v)\ \text{v:=v/2; u:=uu); v:=v-1; z:=zu}\ \{zu^v = x^y\}$$

Aplicando (R6) A (5), se tiene

$$(6)\ \{zu^v = x^y\}\ \text{(while } (v \neq 0)\ \text{(while } (2 \mid v)\ \text{v:=v/2; u:=uu)}\ \text{v:=v-1; z:=zu)}\ \{0 = v \wedge zu^v = x^y\}$$

Finalmente, aplicando (R3) y (R2) a (1) y (6), se tiene

$$\{0 \leqslant x, y\}\ \textit{potencia}\ \{z = x^y\}$$

Ejemplo 4.10. Sea divReal un programa escrito en C que calcula, con un error fijado e, la división de dos reales a y b.

```
void divReal(){
        float a, b, e, z, zz, y, yy;
        scanf("%f",&a);
        scanf("%f",&b);
        scanf("%f",&e);
        z=0;
        zz=0;
        y=1;
        yy=b;
        while(y>e){
                y=y/2;
                yy=yy/2;
                if(zz+yy<=a){
                        z=z+y;
                        zz=zz+yy;
                }
        }
        printf("a/b␣in␣[%f,␣%f)",z,z+e);
        return;
}
```

Se quiere verificar lo siguiente: (0) Sean a, b y e tres reales no negativos donde e no es cero y b es estrictamente mayor que a. Si divReal termina su ejecución, entonces la división de a y b en los reales está acotada inferiorente por z y estrictamente acotada superiormente por z+e. Para verificar este programa, se demostrará $\{0 \leqslant a < b \wedge 0 < e\} divReal \{z \leqslant \frac{a}{b} < z + e\}$ a través del método de Hoare de la siguiente forma:

Se tiene que el invariante del bucle es $INV \equiv zz \leqslant a \wedge zz = b*z \wedge yy = b*y \wedge a < zz+yy$. A través de las reglas (R1) y (R2), se obtiene

(1) $\{0 \leqslant a \wedge 0 = b*0 \wedge b = b*1 \wedge a < 0+b\}$ z:=0; zz:=0; y:=1; yy:=b; {INV}

(2) $\{zz \leqslant a \wedge zz = b*z \wedge yy/2 = b*y/2 \wedge a < zz + 2yy/2\}$ y:=y/2; yy:=yy/2; $\{zz \leqslant a \wedge zz = b*z \wedge yy = b*y \wedge a < zz + 2yy\}$

(3) $\{zz \leqslant a \wedge zz = b*z \wedge yy = b*y \wedge a < zz + yy + yy \wedge zz + yy \leqslant a\}$ z:=z+y; zz:=zz+yy {INV}

Como

(4) $(zz \leqslant a \wedge zz = b*z \wedge yy = b*y \wedge a < zz + yy + yy \wedge zz + yy > a) \rightarrow$ INV,

se puede aplicar la regla condicional simple a (3) y (4) y obtener

(5) $\{zz \leqslant a \wedge zz = b*z \wedge yy = b*y \wedge a < zz + 2yy\}$ if $(zz + yy \leqslant a)$ z:=z+y; zz:=zz+yy {INV}

Aplicando (R2) a (2) y (5), se obtiene el cuerpo del bucle

(6) $\{zz \leqslant a \wedge zz = b*z \wedge yy/2 = b*y/2 \wedge a < zz + 2yy/2\}$ y:=y/2; yy:=yy/2; if $(zz + yy \leqslant a)$ z:=z+y; zz:=zz+yy {INV}

Como

$$(7)\ \text{INV} \wedge \neg(y \leqslant e)) \rightarrow \text{INV}$$

Aplicando (R3) a (6) y (7), se obtiene

$$(8)\ \{\text{INV} \wedge \neg(y \leqslant e)\}\ \text{y:=y/2; yy:=yy/2; if } (zz + yy \leqslant a)\ \text{z:=z+y; zz:=zz+yy } \{\text{INV}\}$$

Al usar la regla iterativa (R6) en (8), queda

$$(9)\ \{\text{INV}\}\ (\text{while } (\neg y \leqslant e)\ \text{y:=y/2; yy:=yy/2; if } (zz + yy \leqslant a)\ \text{z:=z+y; zz:=zz+yy }) \{\text{INV} \wedge y \leqslant e\}$$

Por lo tanto, al aplicar (R2) a (1) y (9), se obtiene

$$(10)\ \{0 \leqslant a \wedge 0 = b * 0 \wedge b = b * 1 \wedge a < 0 + b\}\ \text{z:=0; zz:=0; y:=1; yy:=b;}$$
$$(\text{while } (\neg y \leqslant e)\ \text{y:=y/2; yy:=yy/2; if } (zz + yy \leqslant a)\ \text{z:=z+y; zz:=zz+yy})\ \{\text{INV} \wedge y \leqslant e\}$$

y como se tienen las siguientes condiciones

$$(11)\ (0 \leqslant a < b \wedge 0 < e) \rightarrow (0 \leqslant a \wedge 0 = b * 0 \wedge b = b * 1 \wedge a < 0 + b)$$

y

$$(12)\ (zz \leqslant a \wedge zz = b * z \wedge yy = b * y \wedge a < zz + yy \wedge y \leqslant e) \rightarrow (z \leqslant \frac{a}{b} < z + e)$$

pues si se cumple INV $\wedge\, y \leqslant e$, se tiene que $z = \frac{zz}{b} \leqslant \frac{a}{b} < \frac{zz+yy}{b} = \frac{b\,z+b\,y}{b} = z + y \leqslant z + e$.

Aplicando (R3) a (10), con (11) y (12), se obtiene

$$(13)\ \{0 \leqslant a < b \wedge 0 < e\} divReal \{z \leqslant \frac{a}{b} < z + e\}$$

Cabe remarcar que la regla de la consecuencia (R3) y la regla condicional simple (R4) fuerzan a incluir aserciones entre las fórmulas de la lógica de Hoare.

4.3. Corrección de H

En este apartado se adaptarán y desarrollarán los conceptos de la corrección de H [1].

Sea H el sistema obtenido a partir de las reglas (R0)-(R6). Se le atribuirán las fórmulas de Th(A) para alguna S-estructura A.

Notación 4.11. Sea $\Phi \subseteq L(S)$ un conjunto de fórmulas. Se escribe $\Phi \vdash_H \{\varphi\}P\{\psi\}$ para indicar que existe una demostración de $\{\varphi\}P\{\psi\}$ en H a partir de las fórmulas de Φ.

En el ejemplo anterior 4.10, se puede ver $\{(4), (7), (11), (12)\} \vdash_H (13)$. El objetivo de esta demostración es poder interpretar (13) como (0). Para poder hacerlo, es necesario introducir primero la noción de **verdad** de un programa asertado en una S-estructura A de un lenguaje L(S).

Sea A una S-estructura con dominio D no vacío.

Definición 4.12. Un **A-estado** es una función δ que asigna a cada variable x un valor del dominio D de A, $\delta : VAR \to D$. Se utilizarán las letras δ, τ para denotar los estados.

Definición 4.13. Se dice que en una S-estructura A, se verifica $\varphi \in L(S)$ para un A-estado δ si $(A, \delta) \models \varphi$, normalmente escrito como $A \models \varphi(\delta)$.

Se define S(A) como el conjunto de los A-estados y se define $\Delta_A(\varphi) := \{\delta : \delta \in S(A)$ tal que $A \models \varphi(\delta)\}$.

Definición 4.14. $\varphi \in L(S)$ **es verdad en A**, escrito $A \models \varphi$, si para todo A-estado δ, se tiene que $A \models \varphi(\delta)$ se verifica.

Si un programa $P \in \mathscr{P}(S)$, A es una S-estructura y S(A) es el conjunto de los A-estados, se define la **función de ejecución** como la función parcial $M_A^P : S(A) \to S(A)$ que determine el **significado** del programa P en la estructura A de la siguiente manera: $M_A^P(\delta) = \tau$, significa que si se considera la ejecución del programa P en la estructura A, entonces si $\delta(v_i)$ es el valor inicial de v_i y P termina su ejecución, se tiene que $\tau(v_i)$ es el valor final de v_i.

Una vez aclarado el concepto de función de ejecución, se puede dar una definición formal de la corrección parcial.

Definición 4.15. Se dice que un programa asertado $\{\varphi\}P\{\psi\}$ es **verdadero en** A, $A \models \{\varphi\}P\{\psi\}$, si para todos los estados $\delta, \tau \in S(A)$, si $A \models \varphi(\delta)$ y $M_A^P(\delta) = \tau$, entonces $A \models \psi(\tau)$.

Un programa asertado es **válido** si es verdadero en toda estructura A.

Observación 4.16. Es necesario remarcar que, aunque $\{\varphi\}P\{\psi\}$ sea verdadero en A, no hay garantía de que P termine. Es por ello que, para dejar margen a la no terminación, se define M_A^P como una función parcial de estado a estado. Además, la terminación del programa P depende de la estructura A.

Definición 4.17. Una regla de demostración es **correcta** si para toda S-estructuras A, si las premisas de las premisas de la regla son verdaderas en A, entonces la conclusión de la regla es verdadera en A.

Proposición 4.18. *Las reglas de demostración de H son correctas.*

Demostración:

(R0) es correcta si $\{\varphi\}\{\varphi\}$ es verdadera en toda estructura A. Es verdadera en A si sean $\delta, \tau \in S(A)$ los estados tales que $A \models \varphi(\delta)$ y $M_A^P(\delta) = \tau$, entonces $A \models \varphi(\tau)$. Como el programa P es el programa vacío, entonces para toda δ se tiene que $M_A^P(\delta) = \delta$, es decir, que $\delta = \tau$. Por lo tanto como $A \models \varphi(\delta)$, entonces $A \models \varphi(\tau)$.

(R1) es correcta si $\{\varphi\binom{t}{x}\}$ x:=t $\{\varphi\}$ es verdadera en toda estructura A. Es verdadera en A si $\delta, \tau \in S(A)$ son tales que $A \models \varphi\binom{t}{x}(\delta)$ y $M_A^P(\delta) = \tau$, entonces $A \models \varphi(\tau)$. Como el programa P es x:=t, entonces para toda δ se tiene que $M_A^{x:=t}(\delta) = \delta\binom{t^A}{x}$, donde $\delta\binom{t^A}{x}$ es un estado definido igual que δ, salvo la variable x que tiene como imagen t^A. Como $A \models \varphi\binom{t}{x}(\delta)$, entonces $A \models \varphi(\delta\binom{t^A}{x})$.

(R2) es correcta si para toda estructura A el programa asertado $\{\varphi\}(P_1; P_2)\{\psi\}$ preserva la veracidad en A de los programas asertados $\{\varphi\}P_1\{\chi\}$ y $\{\chi\}P_2\{\psi\}$.

Supóngase que $\{\varphi\}P_1\{\chi\}$ y $\{\chi\}P_2\{\psi\}$ son verdaderos en A. Entonces para todos los estados $\delta_1, \tau_1 \in S(A)$ tales que $A \models \varphi(\delta_1)$ y $M_A^{P_1}(\delta_1) = \tau_1$, se tiene que $A \models \chi(\tau_1)$, y para todos los estados $\delta_2, \tau_2 \in S(A)$ tales que $A \models \chi(\delta_2)$ y $M_A^{P_2}(\delta_2) = \tau_2$, se tiene que $A \models \psi(\tau_2)$.

$\{\varphi\}(P_1; P_2)\{\psi\}$ será verdadero en A si para $\delta, \tau \in S(A)$ estados tales que $A \models \varphi(\delta)$ y $M_A^{P_1;P_2}(\delta) = \tau$, se tiene que $A \models \psi(\tau)$.

Sea τ_1 el estado tal que $M_A^{P_1}(\delta) = \tau_1$. Como $\{\varphi\}P_1\{\chi\}$ es verdadero en A, entonces $A \models \chi(\tau_1)$. Como $M_A^{P_1;P_2}(\delta) = \tau$ es igual a $M_A^{P_2}(M_A^{P_1}(\delta)) = \tau$, entonces se tiene que $M_A^{P_2}(\tau_1) = \tau$. Como $A \models \chi(\tau_1)$ y $\{\chi\}P_2\{\psi\}$ es verdadero, llegamos a que $A \models \psi(\tau)$.

(R3) es correcta si para toda estructura A el programa asertado $\{\varphi\}P\{\psi\}$ preserva la veracidad en A del programa asertado $\{\varphi_1\}P\{\psi_1\}$ y la verificación en A de $(\varphi \to \varphi_1)$ y $(\psi_1 \to \psi)$.

Como se tiene que $(\varphi \to \varphi_1)$ y $(\psi_1 \to \psi)$, entonces se cumple $\Delta_A(\varphi) \subseteq \Delta_A(\varphi_1)$ y $\Delta_A(\psi_1) \subseteq \Delta_A(\psi)$. Supóngase que $\{\varphi_1\}P\{\psi_1\}$ es verdadero en A, entonces para $\delta, \tau \in S(A)$ estados tales que $A \models \varphi_1(\delta)$ y $M_A^P(\delta) = \tau$, se tiene que $A \models \psi_1(\tau)$.

Sean $\delta, \tau \in S(A)$ estados tales que $A \models \varphi(\delta)$ y $M_A^P(\delta) = \tau$. Como $\delta \in \Delta_A(\varphi)$ y $\Delta_A(\varphi) \subseteq \Delta_A(\varphi_1)$, entonces $\delta \in \Delta_A(\varphi_1)$, es decir, $A \models \varphi_1(\delta)$. Para el estado τ tal que $M_A^P(\delta) = \tau$, como $\{\varphi_1\}P\{\psi_1\}$ es verdadero en A se tiene que $A \models \psi_1(\tau)$. Por lo tanto $\tau \in \Delta_A(\psi_1)$, como $\Delta_A(\psi_1) \subseteq \Delta_A(\psi)$, entonces $A \models \psi(\tau)$. Por lo tanto $\{\varphi\}P\{\psi\}$ es verdadero en A.

(R4) es correcta si para toda estructura A el programa asertado $\{\varphi\}$ if (b) P $\{\psi\}$ preserva la veracidad en A del programa asertado $\{\varphi \wedge b\}P\{\psi\}$ y la verificación en A de $(\varphi \wedge \neg b) \to \psi$.

$\{\varphi\}$ if (b) P $\{\psi\}$ es verdadero en A si para $\delta, \tau \in S(A)$ estados tales que $A \models \varphi(\delta)$ y $M_A^{\text{if(b)P}}(\delta) = \tau$, entonces $A \models \psi(\tau)$.

Supóngase $\delta, \tau \in S(A)$ estados tales que $A \models \varphi(\delta)$ y $M_A^{\text{if(b)P}}(\delta) = \tau$, como $\Delta_A(\varphi) = \Delta_A(\varphi \wedge b) \cup \Delta_A(\varphi \wedge \neg b)$ hay dos opciones: o bien $\delta \in \Delta_A(\varphi \wedge b)$ o bien $\delta \in \Delta_A(\varphi \wedge \neg b)$.

Si $\delta \in \Delta_A(\varphi \wedge b)$, entonces $A \models (\varphi \wedge b)(\delta)$ y $\tau = M_A^{\text{if(b)P}}(\delta) = M_A^P(\delta)$. Como $\{\varphi \wedge b\}P\{\psi\}$ es verdadero en A, se llega a $A \models \psi(\tau)$.

Si $\delta \in \Delta_A(\varphi \wedge \neg b)$, como $(\varphi \wedge \neg b) \to \psi$, se tiene $\Delta_A(\varphi \wedge \neg b) \subseteq \Delta_A(\psi)$, entonces $\delta \in \Delta_A(\psi)$. Como $\delta \in \Delta_A(\varphi \wedge \neg b)$ implica que $\tau = M_A^{\text{if(b)P}}(\delta) = M_A(\delta) = \delta$, pues es la operación vacía, se tiene que $\tau \in \Delta_A(\psi)$, que equivale a $A \models \psi(\tau)$.

(R5) es correcta si para toda estructura A el programa asertado $\{\varphi\}P\{\psi\}$ preserva la veracidad en A de los programas asertados $\{\varphi \wedge b\}P_1\{\psi\}$ y $\{\varphi \wedge \neg b\}P_2\{\psi\}$, donde P representa el programa if (b) P_1 else P_2.

$\{\varphi\}P\{\psi\}$ es verdadero en A si para $\delta, \tau \in S(A)$ estados tales que $A \models \varphi(\delta)$ y $M_A^P(\delta) = \tau$, entonces $A \models \psi(\tau)$.

Supóngase $\delta, \tau \in S(A)$ estados tales que $A \models \varphi(\delta)$ y $M_A^P(\delta) = \tau$. Como $\delta \in \Delta_A(\varphi \wedge b) \cup \Delta_A(\varphi \wedge \neg b)$ hay dos opciones, o bien $\delta \in \Delta_A(\varphi \wedge b)$ o bien $\delta \in \Delta_A(\varphi \wedge \neg b)$. Si $\delta \in \Delta_A(\varphi \wedge b)$ se resuelve igual que en (R4) y $\delta \in \Delta_A(\varphi \wedge \neg b)$ es análogo al otro caso,

pero para el programa asertado $\{\varphi \wedge \neg b\}P_2\{\psi\}$.

(R6) es correcta si para toda estructura A el programa asertado $\{\varphi\}W\{\varphi \wedge \neg b\}$ preserva la veracidad en A del programa asertado $\{\varphi \wedge b\}P\{\varphi\}$, donde W representa el programa while (b) P.

$\{\varphi\}W\{\varphi \wedge \neg b\}$ es verdadero en A si para $\delta, \tau \in S(A)$ estados tales que $A \models \varphi(\delta)$ y $M_A^W(\delta) = \tau$, entonces $A \models (\varphi \wedge \neg b)(\tau)$. Se demostrará iterando el número de vueltas del bucle.

Supóngase que el programa W hace 0 vueltas antes de terminar su ejecución y que $\delta, \tau \in S(A)$ estados tales que $A \models \varphi(\delta)$ y $M_A^W(\delta) = \tau$. Como no se ha entrado en el bucle, se sabe que $\tau = M_A^W(\delta) = M_A(\delta) = \delta$ y que $\delta \in \Delta_A(\varphi \wedge \neg b)$. Por lo tanto, se obtiene $A \models (\varphi \wedge \neg b)(\tau)$.

Supóngase que se ha demostrado que $\{\varphi\}W\{\varphi \wedge \neg b\}$ es verdad en A cuando W es un bucle de K iteraciones.

Supóngase que el programa W hace K+1 vueltas antes de terminar su ejecución. Como hace al menos una vuelta del bucle, W es equivalente a $(P; W_K)$, donde W_K es el programa W que hace K vueltas. Para $\delta, \tau \in S(A)$ estados tales que $A \models \varphi(\delta)$ y $M_A^W(\delta) = \tau$, como W hace necesariamente al menos una vuelta, entonces $\delta \in \Delta_A(\varphi \wedge b)$ y $\{\varphi \wedge b\}P\{\varphi\}$ y $\{\varphi\}W_K\{\varphi \wedge \neg b\}$ son verdaderos en A. Aplicando (R2), se llega a $A \models (\varphi \wedge \neg b)(\tau)$.

La proposición 4.18 implica el siguiente teorema:

Teorema 4.19. *Sean $\Phi \subset L(S)$, A una S-estructura y σ un programa asertado. Si $\Phi \vdash_H \sigma$ y $A \models \Phi$, entonces $A \models \sigma$.*

Por consiguiente, para cualquier S-estructura A y cualquier programa adertado σ, $Th(A) \vdash_H \sigma \rightarrow A \models \sigma$.

4.4. Incompletitud de H

La cuestión natural que surge tras haber tratado la corrección de H es si el sistema H, además de correcto, es también completo. No obstante, como ya se verá más adelante, el sistema H es incompleto.

Se puede intentar probar que cualquier programa asertado que sea verdadero en una estructura A del lenguaje del programa es deducible en H. No obstante, por lo general, es imposible demostrar este teorema de completitud. Además, el lenguaje de aserciones puede ser incapaz de expresar los invariantes de los bucles.

En esta sección, se demostrará el resultado de la incompletitud de Wand para la lógica de Hoare [12].

Teorema 4.20. *H es incompleto*

Demostración:

Sea $S = \{f^1, P^1, Q^1, R^1\}$, y A la S-estructura $A = (D, f^A, P^A, Q^A, R^A)$ donde:
El dominio D es $\{a_n : n \in \mathbb{N}\} \cup \{b_n : n \in \mathbb{N}\}$, donde $\{a_n : n \in \mathbb{N}\} \cup \{b_n : n \in \mathbb{N}\}$ es un conjunto de elementos distintos dos a dos,

$f(a_n) = a_{n \dot{-} 1}$ y $f(b_n) = b_{n \dot{-} 1}$, donde $x \dot{-} y = x - y$ si $x \geqslant y$ y, en caso contrario, $x \dot{-} y = 0$,
Px sii $x = a_0$,
Qx sii $x = b_0$,
Rx sii $x = a_{\frac{k(k+1)}{2}}$ para algún $k \in \mathbb{N}$.

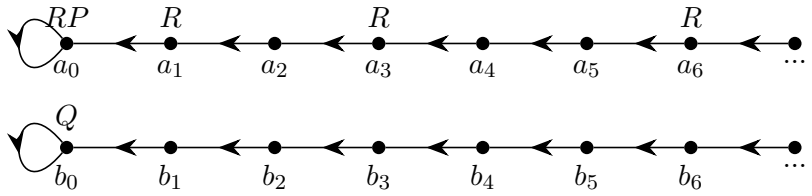

Figura 4: Diagrama de la estructura A, basado en [12].

Para poder demostrar la incompletitud, se utilizará la siguiente definición:

Definición 4.21. Si $\mathscr{A}$ y $\mathscr{B}$ son S-estructuras, un S-isomorfismo $h : \mathscr{A} \to \mathscr{B}$ es una biyección $A \to B$ tal que para cada constante $c \in L(S)$ $h(c^{\mathscr{A}}) = c^{\mathscr{B}}$ para cualquier símbolo de función n-ario $f \in L(S)$ y $a_1, ..., a_n \in A$, $h(f^{\mathscr{A}}(a_1, ..., a_n)) = f^{\mathscr{B}}(h(a_1), ..., h(a_n))$, y para cualquier símbolo de relación R n-ario $R^{\mathscr{A}}(a_1, ..., a_n)$ sii $R^{\mathscr{B}}(h(a_1), ..., h(a_n))$.

Observación 4.22. Si hay un S-isomorfismo $h : \mathscr{A} \to \mathscr{B}$, entonces $Th(\mathscr{A}) = Th(\mathscr{B})$. Si $S_1 \subset S_2$ y $\mathscr{A}$ y $\mathscr{B}$ son S_1-estructuras, entonces $\mathscr{A}$ y $\mathscr{B}$ pueden ser S_1-isomorfas y no S_2-isomorfas.
Un S-isomorfismo $\mathscr{A} \to \mathscr{A}$ se llama S-automorfismo.

Proposición 4.23. *$\{a_n : n \in \mathbb{N}\}$ no es definible en L(S).*

Demostración: Supóngase que existe $\chi \in L(S)$ que define $\{a_n : n \in \mathbb{N}\}$. Es decir, tal que para toda $x \in D$, $A \models \chi(x)$ si y solo si $x \in \{a_n : n \in \mathbb{N}\}$.

Sea $S' = S \cup \{U^1\}$, y $A' = (D, f^{A'}, P^{A'}, Q^{A'}, R^{A'}, U^{A'})$ la S'-estructura donde $f^{A'} = f^A$, $P^{A'} = P^A$, $Q^{A'} = Q^A$, $R^{A'} = R^A$, y $U^{A'}x$ sii $x \in \{a_n : n \in \mathbb{N}\}$.

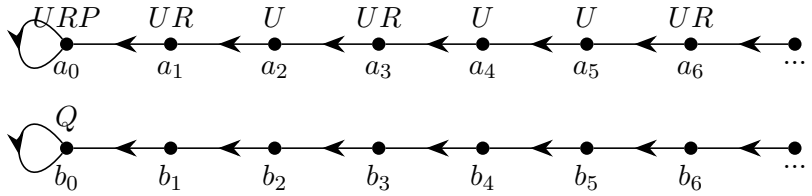

Figura 5: Diagrama de la estructura A', basado en [12].

Entonces

$$A' \models \forall x(\chi(x) \leftrightarrow U(x)),$$

$$Th(A') \models \forall x(\chi(x) \leftrightarrow U(x)).$$

Sean A^+ y B^+ S'-estructuras tales que $A^+, B^+ \models Th(A')$. Si $h : A^+ \to B^+$ es un S-isomorfismo, entonces h preserva U , pues $\chi^{A^+}(x)$ sii $\chi^{B^+}(h(x))$, implica $U^{A^+}(x)$ sii $U^{B^+}(h(x))$ y, por lo tanto, h es un S'-isomorfismo.

Sin embargo, se puede ver que existe un modelo de Th(A') con un S-automorfismo que no es un S'-automorfismo, lo cual contradice la existencia de la fórmula χ.

Sea

$$S'' = S' \cup \{c_n : n \in \mathbb{Z}\} \cup \{d_n : n \in \mathbb{Z}\}$$

y

$$\begin{aligned}\Phi = Th(A') \cup \{&\neg Pc_n \wedge \neg Qc_n \wedge \neg Rc_n \wedge Uc_n \wedge f(c_n) = c_{n\dot{-}1} \mid n \in \mathbb{Z}\} \\ \cup \{&\neg Pd_n \wedge \neg Qd_n \wedge \neg Rd_n \wedge \neg Ud_n \wedge f(d_n) = d_{n\dot{-}1} \mid n \in \mathbb{Z}\}\end{aligned}$$

Cualquier subconjunto finito Φ_0 de Φ se satisface en A' asignándole a los c_n's que aparecen en Φ_0 elementos a_k's, dejando espacio suficiente entre los elementos que cumplen R y asignando los d_n's a elementos b_k. Aplicando el teorema de Compacidad 3.32, Φ es satisfactible y tiene un modelo A".

Se define $h : A'' \to A''$ de la siguiente manera. Si $x \in A''$,

$$h(x) = \begin{cases} d_n^{A''} & si \quad x = c_n^{A''} \text{para algún } n \in \mathbb{Z} \\ c_n^{A''} & si \quad x = d_n^{A''} \text{para algún } n \in \mathbb{N} \\ x & \text{en otro caso} \end{cases}$$

Obsérvese que $h(f(c_n^{A''})) = h(c_{n\dot{-}1}^{A''}) = d_{n\dot{-}1}^{A''} = f(d_n^{A''}) = f(h(c_n^{A''}))$. Y además, como A" es un modelo de Φ, entonces:

$$\begin{aligned} R^{A''}x &\leftrightarrow R^{A''}h(x), \\ Q^{A''}x &\leftrightarrow Q^{A''}h(x), \\ P^{A''}x &\leftrightarrow P^{A''}h(x), \\ h(d_n^{A''}) &= c_n^{A''} \end{aligned}$$

Sin embargo, h no preserva el símbolo de relación U, pues $U^{A''}c_n$, pero $\neg\, U^{A''}d_n$, por lo tanto $U^{A''}c_n \not\leftrightarrow U^{A''}h(c_n)$.

Como se ha visto que h es un S-automorfismo, pero no un S'-automorfismo, queda demostrado 4.23 □

Demostrada la proposición y retomando la demostración de la incompletitud, se observa que $A \models \{Rx\}$ while $(\neg Px \wedge \neg Qx)$ x:=f(x) $\{Px\}$. Si $Th(A) \nvdash_H \{Rx\}$ while $(\neg Px \wedge \neg Qx)$ x:=f(x) $\{Px\}$, entonces se tiene que H es incompleto.

Proposición 4.24. *$Th(A) \nvdash_H \{Rx\}$ while $(\neg Px \wedge \neg Qx)$ x:=f(x) $\{Px\}$*

Demostración: Supóngase que $\{Rx\}$ while $(\neg Px \wedge \neg Qx)$ x:=f(x) $\{Px\}$ es demostrable en H a partir de Th(A). Entonces la demostración tendrá la siguiente forma, donde χ es el invariante del bucle:

$$(1)\ \ (\chi \wedge \neg Px \wedge \neg Qx) \rightarrow \chi\binom{f(x)}{x}$$

$$(2)\ \ \{\chi\binom{f(x)}{x}\}\ \text{x:=f(x)}\ \{\chi\}$$

$$(3)\ \ \{\chi \wedge \neg Px \wedge \neg Qx\}\ \text{x:=f(x)}\ \{\chi\}$$

$$(4)\ \ Rx \rightarrow \chi(x)$$

$$(5)\ \ (\chi(x) \wedge (Px \vee Qx)) \rightarrow Px$$

$$(6)\ \ \{\chi(x)\}\ \text{while}\ (\neg Px \wedge \neg Qx)\ \text{x:=f(x)}\ \{\chi(x) \wedge (Px \vee Qx)\}$$

$$(7)\ \ \{Rx\}\ \text{while}\ (\neg Px \wedge \neg Qx)\ \text{x:=f(x)}\ \{Px\}$$

Se puede observar que las fórmulas (1), (4) y (5) pertenecen a Th(A).

Se demuestra que χ es cierto en a_n $\forall n \in \mathbb{N}$. Si $n \in \mathbb{N}$ y $m \in \mathbb{N}$ son tales que m>n y Ra_m, aplicando (4), se tiene que χ es cierta en a_m. Aplicando (1) m-n veces, se llega a que χ es cierta para a_n.

A continuacción, se demuestra que χ es falso en b_n $\forall n \in \mathbb{N}$. Por (5) se sabe que χ es falsa en b_0, pues como se cumple Qb_0, pero no Px_0, χ debe ser falsa en b_0. Aplicando (1) n veces, se deduce que χ es falsa en b_n.

Por consiguiente, χ define en A al conjunto $\{a_n : n \in w\}$, lo cual es una contradicción por la proposición 4.23. Por ello, queda demostrado que $Th(A) \nvdash_H \{Rx\}$ while $(\neg Px \wedge \neg Qx)$ x:=f(x) $\{Px\}$ y, por lo tanto, también queda demostrado que el sistema H es incompleto.

4.5. Completitud de H en el sentido de Cook

Tal y como aparece en [1], una forma para poder hablar de la completitud del sistema de Hoare sorteando los problemas anteriores es utilizar el concepto de completitud utilizado por Cook.

Definición 4.25. Si S es un vocabulario, A una S-estructura, $\varphi, \psi \in L(S)$ y $P \in \mathscr{P}(S)$, se define

$$\text{post}_A(\varphi, P) := \{\tau \in S(A) : \exists \delta[A \models \varphi(\delta) \wedge M_A^P(\delta) = \tau]\}$$

$$\text{pre}_A(P, \psi) := \{\delta \in S(A) : \forall \tau[M_A^P(\delta) = \tau \rightarrow A \models \psi(\tau)]\}$$

Observación 4.26. Los conjuntos anteriores se caracterizan por las siguientes equivalencias:
$A \models \{\varphi\}P\{\psi\}$ si y solo si $\{\delta : A \models \varphi(\delta)\} \subset \text{pre}_A(P, \psi)$ si y solo si $\text{post}_A(\varphi, P) \subset \{\delta : A \models \psi(\delta)\}$

Definición 4.27. Sea $\mathscr{P}_0$ un conjunto de programas y A una S-estructura. Se dice que un lenguaje L=L(S) es **expresable respecto** a A y $\mathscr{P}_0$ si para toda aserción $\varphi \in L(S)$ y programa $P \in \mathscr{P}_0$ existe una aserción $\psi \in L(s)$ que **define** $\text{post}_A(\varphi, P)$ en L. Es decir, que $\{\delta : A \models \psi(\delta)\} = \text{post}_A(\varphi, P)$. Si A es tal que L sea expresable respecto a A y $\mathscr{P}_0$, entonces A es expresiva y se escribe $A \in \text{Exp}(L, \mathscr{P}_0)$.

Definición 4.28. H es **completo en el sentido de Cook** si para todo vocabulario S y toda S-estructura tal que $A \in \text{Exp}(L(S), \mathscr{P}(S))$ y para todo programa asertado σ, se tiene que si $A \models \sigma$, entonces $Th(A) \vdash_H \sigma$.

Teorema 4.29. *H es completo en el sentido de Cook.*

Demostración:

La demostración se hará por inducción sobre la estructura de los programas. Sea S un vocabulario y A un S-estructura $A \in \text{Exp}(L(S), \mathscr{P}(S))$.

Si $A \models \{\varphi\}x := t\{\psi\}$, entonces necesariamente $A \models \varphi \to \psi\binom{t}{x}$. Por la regla de la asignación, se tiene $Th(A) \vdash_H \{\psi\binom{t}{x}\}x := t\{\psi\}$. Y por la regla de la consecuencia y $A \models \varphi \to \psi\binom{t}{x}$, se tiene $Th(A) \vdash_H \{\varphi\}x := t\{\psi\}$.

Si $A \models \{\varphi\}P_1; P_2\{\psi\}$, entonces $A \models \{\varphi\}P_1\{\chi\}$ y $A \models \{\chi\}P_2\{\psi\}$, donde χ define $\text{post}_A(\varphi, P_1)$. Por la hipótesis de inducción, se tiene $Th(A) \vdash_H \{\varphi\}P_1\{\chi\}$ y $Th(A) \vdash_H \{\chi\}P_2\{\psi\}$. Aplicando la regla de la composición, se tiene $Th(A) \vdash_H \{\varphi\}P_1; P_2\{\psi\}$.

Si $A \models \{\varphi\}$ if (b) P_1 else P_2 $\{\psi\}$, entonces $A \models \{\varphi \wedge b\}P_1\{\psi\}$ y $A \models \{\varphi \wedge \neg b\}P_2\{\psi\}$. Por la hipótesis de inducción, $Th(A) \vdash_H \{\varphi \wedge b\}P_1\{\psi\}$ y $Th(A) \vdash_H \{\varphi \wedge \neg b\}P_2\{\psi\}$. Y por la regla condicional compuesta, $Th(A) \vdash_H \{\varphi\}$ if (b) P_1 else P_2 $\{\psi\}$.

Si $A \models \{\varphi\}$ while (b) P $\{\psi\}$ se debe encontrar χ -un invariante del bucle tal que $A \models \{\chi \wedge b\}P\{\chi\}$, $A \models \varphi \to \chi$ y $A \models \chi \wedge \neg b \to \psi$-, pues, aplicando la hipótesis de inducción, se tendrá que $Th(A) \vdash_H \{\varphi\}$ while (b) P $\{\psi\}$. Pues si $A \models \{\chi \wedge b\}P\{\chi\}$, por la hipótesis de inducción, se tiene $Th(A) \vdash_H \{\chi \wedge b\}P\{\chi\}$. Y por lo tanto, usando (R6), se tiene $Th(A) \vdash_H \{\chi\}$ while (b) P $\{\chi \wedge \neg b\}$. Como además $A \models \varphi \to \chi$ y $A \models \chi \wedge \neg b \to \psi$, se deduce a través de (R3) que $Th(A) \vdash_H \{\varphi\}$ while (b) P $\{\psi\}$.

Queda por demostrar que el invariante χ existe.

Sea $C = \{\delta : \exists k, \delta_0, ..., \delta_k[\delta = \delta_k \wedge A \models \varphi(\delta_0) \wedge \forall i < k\,[M_A^P(\delta_i) = \delta_{i+1} \wedge A \models b(\delta_i)]]\}$. Se observa que $\delta \in C$ si y solo si existe un cómputo tal que empiece en un estado que satisface φ y que alcanza el estado δ tras un número finito de vueltas en el bucle. Se demuestra que C es definible en A. Sean $y_1, y_2, ..., y_n$ las variables libres en φ, b, P ó ψ. Y sea $\chi_1 \in L(S)$ la aserción que define $\text{post}_A(\ \varphi, \text{while}\ (\ b \wedge (\neg(y_1 = z_1) \vee ... \vee \neg(y_n = z_n))\ )\ \text{P}\ )$, donde $z_1, ..., z_n$ son nuevas variables. Por la definición de C, si $A \models \exists z_1, ..., z_n\chi_1(\delta)$, entonces $\delta \in C$. Por otro lado, si $\delta \in C$, entonces $A \models \exists z_1, ..., z_n\chi_1(\delta)$, donde los valores elegidos

de z_i son $\delta(y_i)$ para i=1,...,n. Por lo tanto, la fórmula $\exists z_1, ..., z_n \chi_1$ define C en A y es el invariante χ que se buscaba.

4.6. Refinamientos del teorema de Cook

El contenido de este apartado se ha obtenido principalmente de [11].

Definición 4.30. Sea S un vocabulario. Una **S-especificación** es un conjunto de fórmulas $Ax \subseteq L(S)$, tal que todas las variables de las fórmulas están ligadas a cuantificadores universales.

Sea Ax una S-especificación, A una S-estructura y $\phi \in L(S)$.

Definición 4.31. A es un **modelo de de Ax**, $A \models Ax$, si $A \models \phi$ para todo $\phi \in Ax$.

ϕ es una **consecuencia lógica de Ax**, $Ax \models \phi$, si $A \models Ax$ implica $A \models \phi$ para cualquier S-estructura A.

Definición 4.32. La teoría de la S-especificación Ax es Th(Ax) := $\{\phi : \phi$ es una S-sentencia $\wedge\, Ax \models \phi\}$

Definición 4.33. Sea S un vocabulario. PCA(S) := $\{\, \{\varphi\}P\{\psi\} : \varphi, \psi \in L(S), P \in \mathscr{P}(S) \,\}$.

Definición 4.34. Un programa asertado σ es una consecuencia lógica de Ax, $Ax \models \sigma$, si σ es verdadero para todo modelo de Ax.

Definición 4.35. Se definen como teoría de la corrección parcial de A y teoría de la corrección parcial de Ax a los conjuntos

$$\text{HTh(A)} := \{\, \sigma : \sigma \in \text{PCA(S)} \ \wedge A \models \sigma \,\}$$

$$\text{HTh(Ax)} := \{\, \sigma : \sigma \in \text{PCA(S)} \ \wedge Ax \models \sigma \,\}$$

Definición 4.36. Dr(Ax) := $\{\phi \in L(S) : Ax \vdash \phi\}$ es el conjunto de fórmulas formalmente derivables a partir de Ax. Se define como teoría de Hoare a

$$\text{HDr(Ax)} := \{\, \sigma : \sigma \in \text{PCA(S)} \ \wedge Ax \vdash_H \sigma \,\},$$

que corresponde al conjunto de programas asertados derivables a partir de Ax en la lógica de Hoare, es decir, usando las fórmulas de Dr(Ax) como premisas.

Notación 4.37. HDr(A) representa HDr(Th(A)).

Por el teorema de completitud, se sabe que Dr(Ax)=Th(Ax). Y por la corrección de H, también se sabe que HDr(Ax) $\subseteq$ HTh(Ax).

Definición 4.38. La lógica de Hoare es **relativamente completa con respecto a Ax**, cuando HDr(Ax) = HTh(Ax).

Definición 4.39. La lógica de Hoare es **relativamente completa con respecto A**, cuando
HDr(A) = HTh(A).

Teorema 4.40. ***Teorema de Bergstra y Tucker*** *[3][12]:*
Sean A_1, A_2 S-estructuras y $Ax_1, Ax_2 \subseteq L(S)$ S-especificaciónes. Entonces HTh(A_1)=HTh(A_2) si y solo si Th(A_1)=Th(A_2), y HTh(Ax_1)=HTh(Ax_2) si y solo si Th(Ax_1)=Th(Ax_2).

Por tanto, HTh(A) = HTh(Th(A)), es decir, la teoría de la corrección parcial de una S-estructura A es idéntica a la teoría de la corrección parcial de su teoría.

Tal y como se define en 4.27, fijado un vocabulario S, A es una S-estructura **expresiva** si para todo programa $P \in \mathscr{P}(S)$, $\mathrm{post}_A(\varphi, P)$ es definible en A para toda aserción $\varphi \in L(S)$. No obstante, también se puede definir requiriendo la definibilidad de $\mathrm{pre}_A(P, \psi)$ para toda aserción $\psi \in L(S)$.

El teorema de completitud de Cook 4.29 demuestra que la lógica de Hoare es relativamente completa con respecto a cualquier estructura expresiva. A continuación, se darán las bases para un refinamiento de este teorema.

Sea S un vocabulario, A una S-estructura, $P_1, P_2, P \in \mathscr{P}(S)$, $\varphi, \psi, \chi \in L(S)$, b una S-fórmula libre de cuantificadores y $\delta, \tau, \xi \in S(A)$.

Definición 4.41. Un conjunto de estados $E \subseteq S(A)$ es una **propiedad intermedia** para $\{\varphi\}P_1; P_2\{\psi\}$ si se cumple $(M_A^{P_1}(\delta) = \tau \wedge A \models \varphi(\delta)) \rightarrow \tau \in E$ y $(M_A^{P_2}(\tau) = \xi \wedge \tau \in E) \rightarrow A \models \psi(\xi)$.

Definición 4.42. Un conjunto de estados $E \subseteq S(A)$ es una **propiedad ivariante** para $\{\varphi\}$ while (b) P $\{\psi\}$ si $(A \models \varphi(\delta) \rightarrow \delta \in E)$, $(M_A^P(\delta) = \tau \wedge \delta \in E \wedge A \models b(\delta)) \rightarrow \tau \in E$ y $(\tau \in E \wedge A \not\models b(\tau)) \rightarrow A \models \psi(\tau)$.

Definición 4.43. Una S-especificación **Ax es débilmente expresiva** si cumple las siguientes condiciones:

- Para cualquier $\{\varphi\}\, P_1; P_2\, \{\psi\} \in \mathrm{HTh(Ax)}$ hay una fórmula $\chi \in L(S)$ definiendo una propiedad intermedia en cada modelo A de Ax.
- Para cualquier $\{\varphi\}$ while (b) P $\{\psi\} \in \mathrm{HTh(Ax)}$ hay una fórmula $\chi \in L(S)$ definiendo una propiedad invariante en cada modelo A de Ax.

Definición 4.44. Una S-estructura **A es débilmente expresiva** si Th(A) es débilmente expresiva.

Teorema 4.45. *La lógica de Hoare es relativamente completa con respecto a una S-especificación Ax si y solo si Ax es débilmente expresiva. Y en particular, es relativamente completa con respecto a una S-estructura A si y solo si A es débilmente expresiva.*

Proposición 4.46. *Dada cualquier especificación $Ax \subseteq L(S)$ se cumple*

- *$Ax \vdash \{\varphi\}\ x{:=}t\ \{\psi\}$ si y solo si $Ax \vdash (\varphi \rightarrow \psi\binom{t}{x})$, donde $\varphi, \psi \in L(S)$, x aparece libre en φ y t es un S-término.*
- *$Ax \vdash \{\varphi\}\, P_1; P_2\, \{\psi\}$ si y solo si para alguna aserción intermedia $\chi \in L(S)$ $Ax \vdash \{\varphi\}\, P_1\, \{\chi\}$ y $Ax \vdash \{\chi\}\, P_2\, \{\psi\}$, donde $\varphi, \psi \in L(S)$ y $P_1, P_2 \in \mathscr{P}(S)$.*
- *$Ax \vdash \{\varphi\}$ if (b) P_1 else P_2 $\{\psi\}$ si y solo si $Ax \vdash \{\varphi \wedge b\}\, P_1\, \{\psi\}$ y $Ax \vdash \{\varphi \wedge \neg b\}\, P_2\, \{\psi\}$, donde $\varphi, \psi \in L(S)$, $P_1, P_2 \in \mathscr{P}(S)$ y b una S-fórmula libre de cuantificadores.*

- *$Ax \vdash \{\varphi\}$ while (b) $P\,\{\psi\}$ si y solo si para alguna aserción invariante $\chi \in L(S)$ $Ax \vdash (\varphi \to \chi)$, $Ax \vdash \{\chi \wedge b\}\,P\,\{\chi\}$ y $Ax \vdash ((\chi \wedge \neg b) \to \psi)$, donde $\varphi, \psi \in L(S)$, $P \in \mathscr{P}(S)$ y b una S-fórmula libre de cuantificadores.*

Indicación de la demostración del teorema 4.45:

Para la primera implicación, se asume que Ax es débilmente expresiva. Para demostrar que H es relativamente completa con respecto a Ax, como se sabe que HDr(Ax) $\subseteq$ HTh(Ax), basta con demostrar que HTh(Ax) $\subseteq$ HDr(Ax). Se asume que $\{\varphi\}\,P\,\{\psi\} \in$ HTh(Ax) y se usa la proposición 4.46 y la inducción sobre P para verificar que $\{\varphi\}\,P\,\{\psi\} \in$ HDr(Ax). Si P es una asignación, es trivial. Si P es un condicional, basta con aplicar la hipótesis de inducción a los componentes de P. Y si P es una composición o una iteración, como Ax es débilmente expresiva, se aplican sus condiciones y la hipótesis de inducción.

Para la segunda implicación, se asume que HTh(Ax)=HDr(Ax) y que $\{\varphi\}\,P\,\{\psi\} \in$ HDr(Ax), donde P es una composición o una iteración. Por la proposición 4.46, si P es una descomposición de la forma $(P_1; P_2)$, hay una aserción intermedia $\chi \in L(S)$ tal que $AX \vdash \{\varphi\}\,P_1\,\{\chi\}$ y $AX \vdash \{\chi\}\,P_1\,\{\psi\}$, si P es una iteración de la forma (while (b) P), hay una aserción invariante $\chi \in L(S)$ tal que $AX \vdash (\varphi \to \chi)$, $Ax \vdash \{\chi \wedge b\}\,P\,\{\chi\}$ y $Ax \vdash \{\chi \wedge \neg b\}\,P\,\{\psi\}$. Como la lógica de Hoare es correcta, χ define la propiedad intermedia o invariante en cada modelo de Ax y, por tanto, Ax es débilmente expresiva.

5. Métodos para la verificación de terminación de programas

La verificación de terminación de programas iterativos no es fácilmente reducible a esquemas formales. A continuación se presentan dos métodos que lo intentan en cierta medida: la técnica de Floyd y la técnica de los contadores propuesta por Luckhamm y Suzuki [6].

Observación 5.1. Ni el método de Floyd ni el método de los contadores se pueden usar para estudiar la terminación de todos los programas, tan solo son aplicables en algunos casos. En el apartado 2.3 ya se demostró que no es posible a causa del problema de parada.

Conjuntos Ordenados

Para poder explicar el método de Floyd, es conveniente hacer primero una introducción al concepto de conjunto bien ordenado.

Definición 5.2. Sea X un conjunto no vacío. La relación $\leq$ es una **relación de orden** en X si cumple las siguientes propiedades:

- Reflexiva: $x \leq x, \forall x \in X$
- Antisimétrica: Si $x \leq y$ e $y \leq x$, entonces $x = y$.
- Transitiva: Si $x, y, z \in X$ son tales que $x \leq y$ e $y \leq z$, entonces $x \leq z$.

Si $\leq$ es una relación de orden en un conjunto no vacío X, entonces (X,$\leq$) es un **conjunto ordenado**.

Definición 5.3. Se dice que un conjunto ordenado (X,$\leq$) está **totalmente ordenado**, o que $\leq$ es total, si todo par de elementos de X es comparable a través de la relación, es decir, cuando $x \leq y$ ó $y \leq x$, $\forall x, y \in X$

Definición 5.4. Se dice que (X,$\leq$) está **bien ordenado**, o también que $\leq$ es un buen orden en X, si todo subconjunto no vacío M de X tiene elemento mínimo.

Observación 5.5. Si (X,$\leq$) es un conjunto bien ordenado, no existe ninguna sucesión infinita $\{x_n\}_{n\in\mathbb{N}}$ tal que $x_n \in X$ e $x_{n+1} < x_n$, $\forall n \in \mathbb{N}$.

Proposición 5.6. *Si (X,$\leq$) es un conjunto bien ordenado, entonces (X,$\leq$) es un conjunto totalmente ordenado.*

Demostración: (X,$\leq$) es un conjunto totalmente ordenado si para cualquier $x, y \in X$, x e y son comparables.

Considérese C=$\{x, y\} \subset X$ un conjunto no vacío. Como (X,$\leq$) es un conjunto bien ordenado, entonces C tiene un mínimo m en (X,$\leq$). Como m es un elemento de C, ha de ser m=x ó m=y. Si m=x, entonces $x \leq y$. Si m=y, entonces $y \leq x$. Por tanto, en ambos casos x e y son comparables y, por consiguiente, (X,$\leq$) es totalmente ordenado.

Observación 5.7. El recíproco de la proposición anterior no es cierto; que (X,$\leq$) sea totalmente ordenado no implica que sea un conjunto bien ordenado. Por ejemplo, el orden usual en $\mathbb{R}$ es total y no es un buen orden.

5.1. Método de Floyd para la terminación

Esta técnica consiste en lo siguiente:

(1) Definir un dominio B con una relación de orden $\leqslant$, que sea un buen orden en B, pues garantiza que no existan sucesiones decrecientes infinitas.

(2) Definir, en los puntos A de paso repetido de un ciclo del programa, aplicaciones de la forma $f : D^n \to B$, donde D es el conjunto de estados del programa (ver definición 4.12), de tal manera que para cada estado de la información X_A al pasar por el punto A está definida su imagen en B $f(X_A)$.

(3) El proceso es tal que cada vez que se pasa por el punto A ocurre $f(X_{A1}) > f(X_{A2})$, donde X_{A1} y X_{A2} son los estados correspondientes a pasos consecutivos por el punto A.

Por tanto, para aplicar este método, es necesario encontrar un dominio B con un buen orden y una función f de forma que pueda demostrarse la propiedad (3). En ese caso, cada vez que se repite el ciclo y se pasa por el punto A, se genera un término de la sucesión:

$$f(X_{A1}) > f(X_{A2}) > ... > f(X_{An}) > ...$$

Como (B,$\leqslant$) es un conjunto bien ordenado, no puede ser una sucesión infinita y, por lo tanto, el proceso necesariamente terminará.

Sea $A(V_1)$ la condición previa a una iteración, $A(V_2)$ la misma condición tras la iteración formulada en el mismo punto, y $P(V_1, V_2)$ la condición representativa del efecto del tramo de programa entre el estado inicial V_1 y final V_2 de un paso por el bucle.

Para la verificación parcial, basta demostrar como válida

$$(A(V_1) \wedge P(V_1, V_2)) \to A(V_2) \tag{5.1}$$

Para la verificación de la terminación basta con demostrar,

$$(A(V_1) \wedge P(V_1, V_2)) \to f(V_1) > f(V_2) \tag{5.2}$$

Observación 5.8. La función f puede ser parcial, en ese caso solo los elementos de un subconjunto de D^n pueden tener imagen mediante f. Cuando esto ocurre, la estructura del proceso dentro de la iteración debe garantizar que los estados resultantes pertenecen al subconjunto y por lo tanto tiene imagen mediante f.

Si el subconjunto de D^n con imagen por f se define por la condición C(X), entonces el invariante en A debe implicar C, es decir, $I_A(X) \to C(X)$ debe ser válida.

Si se denomina $R(V_1, V_2)$ al predicado $f(V_1) > f(V_2)$, la fórmula 5.2 es formalmente demostrable en el cálculo de predicados, introduciendo una axiomática descriptiva de las propiedades de f y la relación $\leqslant$.

Ejemplo 5.9. Verificar la terminación con el método de Floyd del programa de la función de McCarthy 91.

El programa representado a continuación en un diagrama de flujo calcula la función:

$$M(x) = \begin{cases} x - 10 & si \quad x > 100 \\ \\ M(M(x+11)) & si \quad x \leqslant 100 \end{cases}$$

Se puede demostrar que M(x)=91 para toda $x \leqslant 100$.

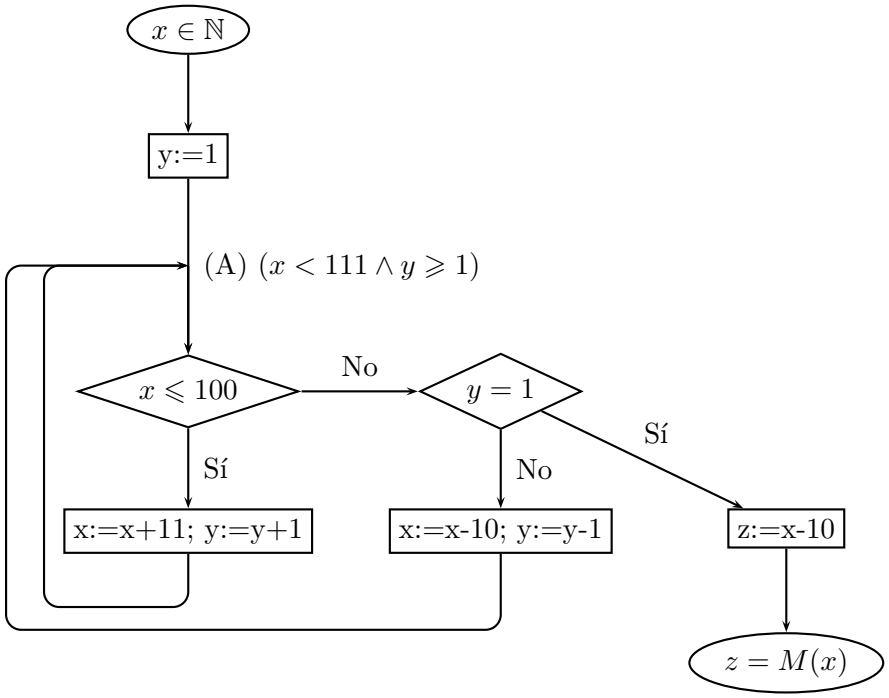

Figura 6: Diagrama de flujo del programa McCarthy 91 (corregido), adaptado de [6].

Se elige como dominio el conjunto bien ordenado de los naturales $\mathbb{N}$ con la relación usual $\leqslant$.

El vector de estado está construido por (x,y), ambos enteros, por lo que el dominio de referencia es I^2, donde I es el conjunto de enteros. Por lo tanto, es preciso definir $f : I^2 \to \mathbb{N}$ tal que si (x_1, y_1) es el estado en A en una iteración y (x_2, y_2) es el estado en A en el paso siguiente, entonces $f(x_1, y_1) > f(x_2, y_2)$.

Para conseguir f, se supondrá que es lineal en x e y: $f(x, y) = ax + by + c$. Y que satisface $ax_1 + by_1 + c > ax_2 + by_2 + c \ \forall x_1, x_2, y_1, y_2 \in I$ y $ax + by + c \in \mathbb{N} \ \forall x, y \in I$

Como al punto A se puede llegar desde dos caminos distintos, se podrá formular x_2 e y_2 en función de x_1 e y_1 de dos formas distintas. Si es por el camino $x \leqslant 100$, enton-

ces $x_2 = x_1+11$ e $y_2 = y_1+1$. Si es por el camino $y \neq 1$, entonces $x_2 = x_1-10$ e $y_2 = y_1-1$.

Por lo tanto, f debe cumplir que:

$$ax_1 + by_1 + c > a(x_1 + 11) + b(y_1 + 1) + c$$
$$ax_1 + by_1 + c > a(x_1 - 10) + b(y_1 - 1) + c$$

o equivalentemente:

$$11a + b < 0$$
$$-10a - b < 0$$

La función f(x,y)=ax+by+c cumple ambas condiciones si a y b cumplen $a < 0$ y $-10a < b < -11a$. En este ejemplo, se elegirá a=-2 e b=21. Para definir c, primero se debe precisar el invariante del bucle que permite definir el subconjunto de valores posibles de x e y que pueden presentarse en A. Una vez definido este subconjunto de los enteros, se define c de manera que necesariamente -2x+21y+c sea un número natural para cualquiera de sus elementos.

El invariante del bucle en A es $I_A(x, y) : (x \leqslant 111) \wedge (y \geqslant 1)$.

Si el bucle va por el camino de $x \leqslant 100$, la condición de relación entre estados inicial (x_1, y_1) y final (x_2, y_2) es $(x_1 \leqslant 100) \wedge (x_2 = x_1 + 11) \wedge (y_2 = y_1 + 1)$. Por tanto, se debe comprobar que es correcta la estructura deductiva

$$I_A(x_1, y_1), (x_1 \leqslant 100), (x_2 = x_1 + 11), (y_2 = y_1 + 1) \rightarrow I_A(x_2, y_2)$$

Sustituyendo I_A por su formulación y reduciendo las conjunciones de la fórmula a sus componentes, basta con demostrar que es correcta la siguiente deducción:

$$(x_1 \leqslant 111), (y_1 \geqslant 1), (x_1 \leqslant 100), (x_2 = x_1 + 11), (y_2 = y_1 + 1) \rightarrow (x_2 \leqslant 111) \wedge (y_2 \geqslant 1)$$

Se puede hacer la verificación formal de esta fórmula introduciendo como premisas propiedades concretas que sean pertinentes de la relación $\leqslant$ en el dominio de números enteros. En este caso se han adoptado las siguientes propiedades.

$$(T1) \quad \vdash x \leqslant a \rightarrow x + b \leqslant a + b$$

$$(T2) \quad \vdash (\ x \geqslant a \wedge b \geqslant 0\ ) \rightarrow x + b \geqslant a$$

La demostración es inmediata:
Partiendo de las premisas

$$(1) \quad x_1 \leqslant 111$$
$$(2) \quad y_1 \geqslant 1$$
$$(3) \quad x_2 = x_1 + 11$$
$$(4) \quad y_2 = y_1 + 1$$
$$(5) \quad x_1 \leqslant 100$$

se aplica (T1) a (5), obteniendo

$$(6) \quad x_1 \leqslant 100 \rightarrow x_1 + 11 \leqslant 111$$

Como se tiene (5) y (6), por Modus Ponens se infiere

$$(7)\ \ x_1 + 11 \leqslant 111.$$

Aplicando la regla de inferencia de identidad a (3) y (7), se tiene

$$(8)\ \ x_2 \leqslant 111,$$

Al aplicar (T2) a (1)

$$(9)\ \ y_1 \geqslant 1 \rightarrow y_1 + 1 \geqslant 1$$

Se aplica Modus Ponens a (2) y (9), obteniendo

$$(10)\ \ y_1 + 1 \geqslant 1$$

Aplicando la regla de inferencia de identidad a (4) y (10), se tiene

$$(11)\ \ y_2 \geqslant 1,$$

A partir de (8) y (11), se obtiene

$$(12)\ \ (x_2 \leqslant 111) \wedge (y_2 \geqslant 1),$$

concluyendo la demostración de la deducción y comprobando que $(x \leqslant 111) \wedge (y \geqslant 1)$ es el invariante del bucle.

Queda ver que el invariante se cumple inicialmente, a partir de la precondición del proceso. $y \geqslant 1$ se verifica porque se inicializa a 1 y si x vale más de 111 no entraría al ciclo y terminaría, y si entra en el ciclo por el camino $x \leqslant 100$, entonces $x \leqslant 111$.

Por lo tanto, en el bucle del camino $x \leqslant 100$, al pasar por A, el subconjunto de valores de (x,y) se describe por $(x \leqslant 111) \wedge (y \geqslant 1)$.

Queda demostrar que la función f(x,y)=-2x+21y+c definida en este subconjunto tiene imagen en $\mathbb{N}$. El valor mínimo de f en este subconjunto se alcanza para x máximo e y mínimo de los valores posibles. En este caso, esos valores son x=111 e y=1, por lo que f(111,1)=-201+c es el valor mínimo. Para que la función, en el subconjunto, tenga imagen en los naturales, todos sus valores deben ser positivos. Este requisito se cumple imponiendo que f(111,1)>0 y por tanto c>201.

Si el bucle va por el camino de $y \neq 1$, se demuestra que la invariancia $(x \leqslant 111) \wedge (y \geqslant 1)$ es correcta si se demuestra que es correcta la deducción

$$(x_1 \leqslant 111), (y_1 \geqslant 1), (x_1 > 100), (y_1 \neq 1), (x_2 = x_1 - 10), (y_2 = y_1 - 1) \rightarrow (x_2 \leqslant 111) \wedge (y_2 \geqslant 1).$$

Y se puede deducir formalmente si se introduce como premisa (T1) en el dominio de los enteros. Partiendo de las premisas

$$(1)\ \ x_1 \leqslant 111$$
$$(2)\ \ y_1 \geqslant 1$$
$$(3)\ \ x_1 > 100$$
$$(4)\ \ y_1 \neq 1$$
$$(5)\ \ x_2 = x_1 - 10$$
$$(6)\ \ y_2 = y_1 - 1,$$

se aplica (T1) a (1), obteniendo

$$(7)\ \ x_1 \leqslant 111 \rightarrow x_1 - 10 \leqslant 101$$

Por Modus Ponens en (1) y (7), se infiere

$$(8)\ \ x_1 - 10 \leqslant 101$$

Aplicando la regla de inferencia de identidad a (5) y (8), se tiene

$$(9)\ \ x_2 \leqslant 101$$

Como

$$(10)\ \ x_2 \leqslant 101 \rightarrow x_2 \leqslant 111$$

es válida, entonces, por Modus Ponens en (9) y (10), se infiere

$$(11)\ \ x_2 \leqslant 111$$

Como

$$(12)\ \ (y_1 \geqslant 1) \wedge (y_1 \neq 1) \rightarrow y_1 > 1$$

es válida, al aplicarle Modus Ponens con (2) y (4), se infiere

$$(13)\ \ y_1 > 1$$

Como

$$(14)\ \ y_1 > 1 \rightarrow y_1 - 1 \geqslant 1$$

es válida, entonces, por Modus Ponens en (13) y (14), se tiene

$$(15)\ \ y_1 - 1 \geqslant 1$$

Aplicando a (15) y (6) la regla de inferencia de identidad, se tiene

$$(16)\ \ y_2 \geqslant 1$$

Obteniendo, finalmente, a partir de (9) y (16),

$$(17)\ \ (x_2 \leqslant 111) \wedge (y_2 \geqslant 1)$$

Para calcular c, cabe tener en consideración que puede pasarse por A en el bucle de $y \neq 1$ con x>111 para valores iniciales de los que se saldría inicialmente del bucle. En estos casos siendo c>201, podría ocurrir que -2x+21y+c<0 si $c < 2x_{inicial} - 21$. Por lo tanto c debe ser mayor que 201 y que $2x_{inicial} - 21$.

Queda demostrado, pues, que para cualquier entrada el programa termina en un número finito de pasos.

En el ejemplo anterior se ha visto que la construcción de la función f puede ser delicada y en algunos casos no sistematizable.

5.2. Método de los contadores

Este método consiste en introducir una variable contador por cada bucle que haya, inicializarlos fuera y modificar su valor una unidad en cada pasada por el mismo punto interno al bucle.

Observación 5.10. Si se demuestra que la variable contador i está acotada, es decir, $i \leqslant L$ donde L es fija, entonces el número de veces que se ejecuta un bucle está acotado y, por lo tanto, este termina.

El límite L puede ser una constante o una expresión independiente del bucle.

La verificación de la terminación con contadores se reduce a demostrar que la estructura del programa garantiza que todo contador de ciclo i está acotado.

La técnica de utilización de contadores es menos formalizable que el método de Floyd. No obstante, es más sencilla y permite evaluar límites superiores del número de iteraciones de un ciclo, lo que resulta útil a efectos de estimación de rendimientos en la ejecución del proceso.

Ejemplo 5.11. Verificar la terminación con el método de los contadores del programa de la función de McCarthy 91.

El programa representado en la figura 7 en la página 42 es una adaptación de la figura 6 para aplicar el método de los contadores.

Los contadores i y j cuentan las veces que se pasan por el bucle del camino $x \leqslant 100$ e $y \neq 1$ respectivamente. El contador n cuenta las veces que el proceso pasa por el punto (A), independientemente del camino del bucle por el que lo haga.

Nótese que en el camino $x \leqslant 100$ se le suma 1 a y en cada pasada y en el camino $y \neq 1$ se le resta 1. Como y se inicializa como y=1, necesariamente y=i-j+1.

Análogamente, x aumenta en 11 y disminuye en 10 en los caminos $x \leqslant 100$ e $y \neq 1$ respectivamente. Por lo tanto, si x_0 es el valor inicial, entonces $x = x_0 + 11i - 10j$.

En el análisis realizado por el método de Floyd en el ejemplo 6, $y \geqslant 1$ es un invariante en A, por lo tanto $i - j + 1 \geqslant 1$, de donde $i \geqslant j$, que implica que el camino $x \leqslant 100$ se ejecutará más veces que el camino $y \neq 1$.

Como claramente n=i+j, si se demuestra que n está acotado superiormente, entonces queda verificada la terminación del programa sin la necesidad de acotar i y j.

A continuación se acotará superiormente n. Como $i, j \geqslant 0$ y como se ha demostrado que $x = x_0 + 11i - 10j$ y que $i \geqslant j$, se pueden encontrar dos límites inferiores de x de la siguiente forma:

$$x = x_0 + 11i - 10j \geqslant x_0 + 11i - 10i = x_0 + i$$
$$x = x_0 + 11i - 10j \geqslant x_0 + 11j - 10j = x_0 + j$$

Sumando estos resultados, se obtiene $2x \geqslant 2x_0 + i + j$, que es equivalente a $2x \geqslant 2x_0 + n$. Para poder acotar n, se deberán estudiar los posibles valores que pueden adoptar x y x_0.

Si $x_0 > 100$, el programa acaba con $x_0 - 10$, pues y=1 y, por lo tanto, n=1. Si $x_0 < 100$, entra en el bucle y, como en el ejemplo 6 se ha demostrado que $x \leqslant 111$ es invariante, entonces $n \leqslant 222 - 2x_0$.
Por lo tanto, como n está acotado superiormente y como la serie de valores de n es creciente, entonces el proceso termina.

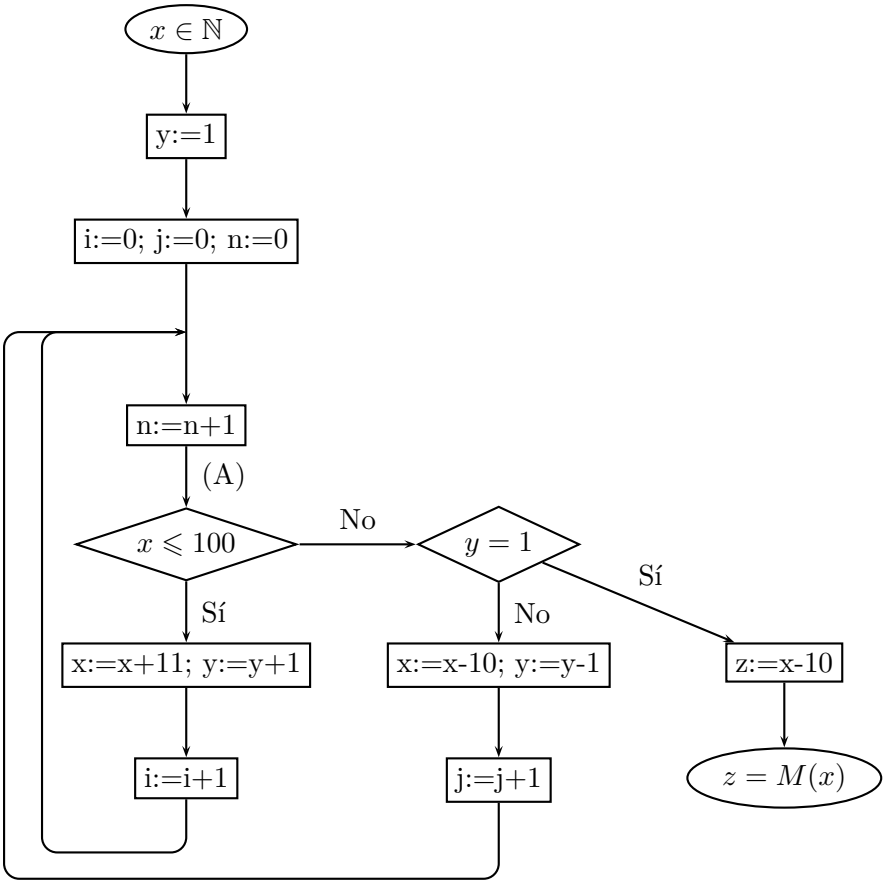

Figura 7: Diagrama de flujo del programa McCarthy 91 adaptado al método de los contadores [6].

6. Conclusiones

Esta memoria ha buscado dar una introducción autocontenida al tema de verificación de programas. Como entre los artículos referenciados variaba la notación y algunas de las definiciones utilizadas, para facilitar la comprensión al lector, se ha adaptado la información obtenida de los documentos para que sea consistente a lo largo de toda la memoria.

Se han mostrado diversos ejemplos de cómo aplicar el método de Floyd y la lógica de Hoare para verificar la corrección parcial. Al querer prestar especial atención a la lógica de Hoare, se han demostrado las cuestiones de corrección, incompletitud y completitud en el sentido de Cook del sistema H. A pesar de haberse fundamentado principalmente en el trabajo de Apt [1], en este trabajo se ha utilizado la demostración de Wand [12] de la incompletitud, en vez de la demostración que utiliza Apt, basándose en [4], pues esta demostración usa conceptos que no se han tratado en esta memoria.

Como no se ha tratado la verificación de la corrección total -la cual indica que un programa termina y es correcto- en el tema de métodos para la verificación de la terminación, se han presentado herramientas para complementar la verificación de la corrección parcial. Por ello, una ampliación natural de este trabajo sería profundizar en el tema de la verificación de programas elaborando el tema de la corrección total.

Otras continuaciones interesantes para esta memoria podrían ser estudiar la verificación de otros programas a parte de los iterativos, como por ejemplo los recursivos y en paralelo, o introducir el tema de verificación automática. En el caso de que querer trabajar el tema de verificación automática, también se tendrían que introducir los lenguajes de lógica temporal.

Bibliografía

[1] Apt K.R. *Ten Years of Hoare's Logic: A Survey—Part I.* ACM Transaction on Programming Languages and Systems 3 (1981) pp. 431-483.

[2] De Bakker J.W., Meertens L.G.L.T. *On the completeness of the inductive assertion method.* Journal of Computer and System Sciences 11 (1975) pp. 323-357.

[3] Bergstra J.A., Tucker J.V. *Expressiveness and the completeness of Hoare's logic.* Journal of Computer and System Sciences 25 (1982) pp. 267-284.

[4] Bergstra J.A., Tucker J.V. *Some natural structures which fail to possess a sound and decidable hoare-like logic for their while-programs.* Theoretical Computer Science 17 (1982) pp. 303-315.

[5] Bisbal Riera Jesús. *Manual de algorítmica.* Editorial UOC, Barcelona, 2009. (Manuales) pp. 51-74.

[6] Cuena J. *Lógica informática.* Alianza, Madrid, 1985. (Alianza informática ; 1) pp. 301-311.

[7] Ebbinghaus H.D., Flum J., Thomas W. *Mathematical Logic.* 3rd ed. Cham: Springer, New York, 2021.

[8] Floyd W. *Assigning meanings to programs* [Internet]. Mathematical Aspects of Computer Science. Schwartz J, editor. Providence, Rhode Island: American Mathematical Society; (1967). (Proceedings of Symposia in Applied Mathematics; vol. 19). Disponible en: `http://www.ams.org/psapm/019`

[9] Hoare C.A.R. *An axiomatic basis for computer programming.* Commun ACM 12 (1969) pp. 576-580.

[10] Martí Oliet Narciso, Segura Díaz C.M., Verdejo A. *Algoritmos correctos y eficientes : diseño razonado ilustrado con ejercicios.* Garceta, Madrid, 2012 pp. 27-113.

[11] Rodriguez Artalejo M. *Some questions about expressiveness and relative completeness in Hoare's logic.* Theoretical Computer Science 39 (1985) pp. 189-206.

[12] Wand M. *A New Incompleteness Result for Hoare's System.* J ACM 25 (1978) pp. 168-175.