**TCYVP**

##Clase 2. Jerarquía de la computabilidad

**Teorema 2.1. Algunas propiedades de clausura de la clase R**

Considerando las operaciones de complemento, intersección, unión y concatenación de lenguajes, se cumple que la clase R es cerrada con respecto a todas ellas.

**Teorema 2.2. Algunas propiedades de clausura de la clase RE**

Considerando las operaciones de intersección, unión y concatenación de lenguajes, se cumple que también la clase RE es cerrada con respecto a ellas. En cambio, a diferencia de la clase R, RE no es cerrada con respecto al complemento.

**Teorema 2.3. R = RE ⋂ CO-RE**

Se prueba fácilmente que R ⊆ RE ⋂ CO-RE. La inclusión R ⊆ RE se cumple por definición. También vale R ⊆ CO-RE porque L ∈ R → LC ∈ R → LC ∈ RE → L ∈ CO-RE.

##Clase 4. Reducciones de problemas

**Definición 4.1: Reducción de problemas**

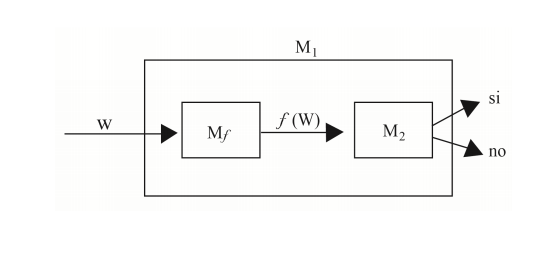
Sean L1 y L2 dos lenguajes incluidos en Ʃ\*. Existe una reducción del lenguaje L1 al lenguaje L2, si y sólo si existe una función total computable f: Ʃ\* → Ʃ\* tal que ∀w ∈ Ʃ\*: w ∈ L1 ↔ f(w) ∈ L2. La función f se denomina función de reducción. Que f sea total computable significa, como se indicó previamente, que existe una MT que a partir de cualquier cadena w computa f(w) en su cinta de salida y se detiene. En general, identificaremos con Mf a la MT que computa f. Que haya una reducción de L1 a L2 significa, entonces, que existe una MT que transforma toda cadena de L1 en una cadena de L2, y toda cadena no perteneciente a L1 en una cadena no perteneciente a L2. Utilizaremos la notación L1 α L2 para expresar que existe una reducción del lenguaje (o problema) L1 al lenguaje (o problema) L2.

**Teorema 4.1. Si L2 está en R (RE) y L1 α L2, entonces L1 está en R (RE)**

Probaremos que si L2 ∈ R y L1 α L2, entonces L1 ∈ R. La prueba de que si L2 ∈ RE y L1 α L2, entonces L1 ∈ RE, es muy similar y queda como ejercicio.

**Idea general.**

Componiendo la MT Mf que computa la función de reducción f del lenguaje L1 al lenguaje L2, con la MT M2 que reconoce el lenguaje L2 y se detiene siempre, se obtiene una MT M1 que reconoce el lenguaje L1 y se detiene siempre.

[](https://camo.githubusercontent.com/e83a9a4af4bb4c8f5e785633f1884cfa769e2fc774ecef2068ca5170c0ba1457/687474703a2f2f692e696d6775722e636f6d2f4f5a4a704670732e706e67)

**Construcción de la MT M1.**

Sea Mf una MT que computa la función de reducción f, con w ∈ L1 ↔ f(w) ∈ L2, y sea M2 una MT que reconoce L2 y se detiene siempre. La MT M1 trabaja de la siguiente manera:

1. Simula Mf a partir de la entrada w y obtiene f(w).
2. Simula M2 a partir de f(w) y acepta si y sólo si M2 acepta.

**Prueba de que M1 se detiene siempre.**

La MT M1 se detiene siempre porque Mf y M2 se detienen siempre.

**Prueba de L1 = L(M1).**

* w ∈ L1 → Mf a partir de w computa f(w) ∈ L2 → M2 a partir de f(w) se detiene en su estado qA → M1 a partir de w se detiene en su estado qA → w ∈ L(M1).
* w ∉ L1 → Mf a partir de w computa f(w) ∉ L2 → M2 a partir de f(w) se detiene en su estado qR → M1 a partir de w se detiene en su estado qR → w ∉ L(M1).

Como corolario del teorema anterior se establece que si L1 no es recursivo y existe una reducción de L1 a L2, entonces L2 tampoco es recursivo (de lo contrario L1 sería recursivo). Lo mismo se puede decir para el caso de los lenguajes recursivamente numerables.

**Ejemplo 4.2. Reducción de HP a LU**

HP = {(, w) | M se detiene a partir de w}, que representa el problema de la detención, no es recursivo.  
LU = {(, w) | M acepta w}, que representa el problema de la pertenencia, tampoco es recursivo.

Una manera alternativa de probar que LU no es recursivo, aplicando el corolario del teorema anterior, es construir directamente una reducción de HP a LU.

**Definición de la función de reducción.**

Si la entrada es sintácticamente válida (el caso inválido se trata después), se define

f((<M>, w)) = (<M’>, w)

tal que M’ se comporta como M, salvo que cuando M se detiene, ya sea en el estado qA o en el estado qR, M’ acepta.

**La función f es total computable.**

Si la entrada no es una cadena válida (, w), la MT Mf genera la cadena 1. En caso contrario, para generar <M’>, Mf modifica las 5-tuplas de reemplazando todo estado qR por el estado qA.

**Se cumple (, w) ∈ HP ↔ f((, w)) ∈ LU.**

(, w) ∈ HP ↔ M se detiene a partir de w ↔ M’ acepta w ↔ (<M’>, w) ∈ LU.

**Ejemplo 4.4. Reducción de LU a LƩ\***

LƩ\* = { | L(M) = Ʃ\*}. Por lo tanto, hay que determinar si LƩ\* es un lenguaje recursivo. Se hará LU α LƩ\*. Como LU ∉ R, entonces LƩ\* ∉ R (si LƩ\* ∈ R, entonces LU ∈ R). Así se probará que el problema de determinar si una MT acepta todas las cadenas de Ʃ\* es indecidible.

**Definición de la función de reducción.**

Para pares válidos (, w) se define f((, w)) = . Donde Mw es una MT que borra su entrada v, la reemplaza por w, simula M a partir de w, y acepta si y sólo si M acepta. Se comprueba fácilmente que:

* Si M acepta w, entonces L(Mw) = Ʃ\*
* Si M no acepta w, entonces L(Mw) = ∅ (es decir que L(Mw) ≠ Ʃ\*, que es lo que se necesita).

**La función f es total computable.**

Si la entrada no es una cadena válida (, w), la MT Mf genera la cadena 1. En caso contrario, para generar , la MT Mf le agrega al código un fragmento inicial que borra la entrada y la reemplaza por w.

**Se cumple (, w) ∈ LU ↔ f((, w)) ∈ LƩ\*.**

(, w) ∈ LU ↔ M acepta w ↔ L(Mw) = Ʃ\* ↔ ∈ LƩ\*.

lenguaje LƩ\* no es ni siquiera recursivamente numerable, como se sugirió recién. Se va a construir una primera reducción de problemas para probar la no pertenencia a la clase RE, lo que suele ser más difícil que las pruebas de no pertenencia a la clase R utilizando reducciones.

**Ejemplo 4.5. Reducción de LU C a LƩ\***

Se probará que LƩ\* = { | L(M) = Ʃ\*} ∉ RE. Se hará LU-Complemento α LƩ.\*. Como LU ∈ RE – R, entonces LU-Complemento ∉ RE, y así, con la reducción propuesta se probará que LƩ\* ∉ RE (si LƩ\* fuera recursivamente numerable también lo sería el lenguaje LU-Complemento).

**Definición de la función de reducción.**

Para pares válidos (, w) se define f((, w)) = . Donde Mw es una MT que a partir de una entrada v simula a lo sumo |v| pasos de M a partir de w (M podría detenerse antes), y acepta si y sólo si M no acepta w. Se comprueba fácilmente que:

* Si M no acepta w, entonces L(Mw) = Ʃ\*
* Si M acepta w, digamos en k pasos, entonces L(Mw) = {v | |v| < k} (es decir que L(Mw) ≠ Ʃ\*, que es lo que se necesita).

**La función f es total computable.**

Si la entrada no es una cadena válida (, w), y establecemos por convención que la misma pertenece a LU-Complemento, entonces Mf genera un código <MƩ\*> tal que L(MƩ\*) = Ʃ\*. En caso contrario, para generar , la MT Mf le agrega al código un fragmento que calcula el tamaño i de la entrada, decrementa i en 1 toda vez que se ejecuta un paso, detiene la ejecución cuando i = 0, y acepta si y sólo si no se alcanza al final el estado qA.

**Se cumple (, w) ∈ LU-Complemento ↔ f((, w)) ∈ LƩ\*.**

(, w) ∈ LU C ↔ M no acepta w ↔ L(Mw) = Ʃ\* ↔ ∈ LƩ\*.

**Clase 6. Jerarquía de la complejidad temporal**

**Definición 6.1: Conceptos básicos de la complejidad temporal**

Dada la función T: N → N, se define que una MT M trabaja en tiempo T(n) si y sólo si para toda entrada w, con |w| = n, M hace a lo sumo T(n) pasos, en su única computación si es determinística, o en cada una de sus computaciones si es no determinística. De modo similar se define una MT que trabaja en tiempo O(T(n)). Se asume que una MT hace siempre al menos n + 1 pasos, para leer toda su entrada.

En efecto, trabajamos con órdenes de magnitud. En lugar de funciones T se utilizan funciones O(T), que se leen “orden de T”. La expresión O(T) denota el conjunto de todas las funciones f que cumplen f(n) ≤ c.T(n), para toda constante c > 0 y todo número natural n ≥ 0.

Los problemas que pueden ser resueltos por MT que trabajan en tiempo O(T(n)) se agrupan en una misma clase: un problema (o lenguaje) pertenece a la clase DTIME(T(n)) (respectivamente NTIME(T(n))) si y sólo si existe una MTD (respectivamente MTN), con una o más cintas, que lo resuelve (o reconoce) en tiempo O(T(n)).

En el conjunto R de los lenguajes recursivos o problemas de decisión decidibles se distinguen dos clases temporales, P y NP, que se definen de la siguiente manera:

P = ⋃i ≥ 0 DTIME(n^i)

NP = ⋃i ≥ 0 NTIME(n^i)

Notar que O(n^i) reúne a todas las funciones polinomiales T(n) = a0 + a1 n + … + ai x n^i. Por lo tanto, la clase P agrupa a los problemas que se resuelven en tiempo determinístico polinomial, y NP es la clase de los problemas que se resuelven en tiempo no determinístico polinomial. Se cumple por definición que P ⊆ NP, y además que las dos clases están incluidas estrictamente en R. En cambio, sólo podemos afirmar que “se sospecha” que P ≠ NP.

**Clase 8. Problemas NP-Completos.**

**Teorema 8.1**

Si L2 está en **P (NP)** y L1 reduce a L2, entonces L1 está en **P (NP)**.

**Corolario 8.1**

Dados dos lenguajes L1 y L2:  
Si L1 está en **P (NP)** y existe una *reduccion polinomial* de L1 a L2,  
entonces L2 tampoco está en **P (NP)**.

Las reducciones son **reflexivas y transitivas** pero no son **simetricas**.

**Definicion 8.2 - Problemas NP-Completos.**

Un lenguaje L0 es NP-*difícil* (o *NP-Hard*, o L0 pertenece a NPH) si y sólo si para todo lenguaje L perteneciente a NP se cumple que L se reduce polinomimialmente a L0. En otras palabras, L0 es NP-dificil si y sólo si para todos los lenguajes de NP se reducen polinomialmente a él (L0).

Ahora, si L0 pertenece a NP, entonces se define que es **NP-Completo** (L0 pertenece a NPC).

**Teorema 8.2: Si un problema NP-completo está en P, entonces P = NP**

La prueba es muy sencilla. Supongamos que un lenguaje NP-completo L0 está en P. Entonces, si L es algún lenguaje de NP:

* Por ser L0 un lenguaje NP-completo, se cumple que L αP L0.
* Y por ser L0 además un lenguaje de P, se cumple que L ∈ P.

Como esto vale para todo lenguaje L de NP, entonces NP ⊆ P y así P = NP.

**Teorema 8.4: Si L1 ∈ NPC, L1 αP L2 , y L2 ∈ NP, entonces L2 ∈ NPC.**

Demostracion: encontrando una reducción polinomial de un lenguaje L1 NP-completo a un lenguaje L2 de NP, se prueba que también L2 es NP-completo. La prueba es muy simple. Sea L algún lenguaje de NP:

* Como L1 ∈ NPC, entonces se cumple L αP L1.
* Como L1 αP L2 , entonces por propiedad transitiva de αP se cumple L αP L2.
* Dado que lo anterior vale para todo lenguale L de NP, entonces L2 es NP-difícil, y como está en NP, también es NP-completo.

**clase 9**

**Definicion NPI**

Asumiendo P ≠ NP, se prueba que NP incluye, además de P y NPC, una tercera subclase de problemas, NPI, así llamada por incluir problemas de dificultad intermedia comparados con los de las otras dos subclases.

La existencia de NPI, asumiendo P ≠ NP, se puede demostrar utilizando un teorema (Teorema de Ladner) que establece que si B es un lenguaje recursivo no perteneciente a P, entonces existe un lenguaje D perteneciente a P, tal que:

* A = D ⋂ B no pertenece a P.
* A se reduce polinomialmente a B.
* B no se reduce polinomialmente a A.

La prueba de este resultado consiste básicamente en construir a partir de B un subconjunto A, extrayendo de B una cantidad de cadenas lo suficientemente grande para que no se cumpla B αP A, y al mismo tiempo no tan grande para que tampoco se cumpla A ∈ P. Aplicando el teorema, la existencia de NPI se demuestra de la siguiente manera:

* Sea B ∈ NPC (asumiendo P ≠ NP, cumple la hipótesis de que está en EXP – P).
* Existe D ∈ P tal que si A = D ⋂ B, entonces A ∉ P, A αP B, y no se cumple B αP A (resultado del teorema).
* Como D ∈ NP y B ∈ NP, entonces D ⋂ B ∈ NP, es decir A ∈ NP.
* Como B ∈ NP y no se cumple B αP A, entonces A ∉ NPC.
* Por lo tanto, A ∈ NP − (P ⋃ NPC), es decir, A ∈ NPI.

**Definicion CO-NP**

CO-NP agrupa, dentro de EXP, a los **lenguajes complemento** de los lenguajes **de NP**.

Enunciamos que P ⊆ NP ⋂ CO-NP, y también destacamos que se conjetura que NP ≠ CO-NP. Notar que la asunción NP ≠ CO-NP es más fuerte que la asunción P ≠ NP: si P = NP, como P es cerrada con respecto al complemento entonces también lo es NP, por lo que vale NP = CO-NP.

Siendo NP la clase de los problemas con certificados suscintos, entonces en CO-NP están los problemas con **descalificaciones suscintas**. Por ejemplo, en el caso de un par válido (G, K) que pertenece a NOCLIQUE, su descalificación suscinta es un clique del grafo G de tamaño K.

Un problema de CO-NP es CO-NP-completo si todos los problemas de CO-NP se reducen polinomialmente a él. Se demuestra fácilmente que si un problema es NP-completo, entonces su complemento es CO-NP-completo.

Un importante representante de los problemas CO-NP-completos es el problema de la validez de las fórmulas booleanas, que consiste en determinar si una fórmula booleana es satisfactible por toda asignación de valores de verdad. Identificando con BVAL al lenguaje que representa este problema, se prueba fácilmente que BVAL es CO-NP-completo:

* BVAL está en CO-NP: toda fórmula booleana de BVAL tiene una descalificación suscinta (una asignación de valores de verdad que no la satisface).
* BVAL es CO-NP-difícil: NOSAT (el problema complemento de SAT) es CO-NP-completo, y se reduce polinomialmente a BVAL mediante la función de reducción f(φ) = ¬φ. Por la transitividad de α P se cumple el enunciado.

Asumiendo NP ≠ CO-NP, se prueba que NPC y NP ⋂ CO-NP son disjuntos.

Formalmente, si L es un lenguaje NP-completo y además está en CO-NP, entonces se cumple que NP = CO-NP. Probamos a continuación el caso NP ⊆ CO-NP, y la demostración de CO-NP ⊆ NP queda como ejercicio (resuelto en la practica):

* Dado un lenguaje L’ perteneciente a NP, se cumple L’ αP L.
* Entonces también se cumple L’Complemento αP L Complemento.
* Como L Complemento ∈ NP, entonces L’ Complemento ∈ NP, o lo que es lo mismo, L’ ∈ CO-NP.

Notar que toda instancia de un problema de NP ⋂ CO-NP tiene o bien un certificado suscinto, cuando la instancia es positiva, o bien una descalificación suscinta, cuando la instancia es negativa.

**9.4 Complejidad Espacial**

**Definiciones básicas**

Para el estudio de la complejidad espacial se utilizan MT con **una cinta de entrada de sólo lectura**, sobre la que el cabezal sólo se mueve a lo largo de la cadena de entrada (más los dos símbolos blancos que la delimitan). El uso de una cinta de entrada de sólo lectura permite trabajar con orden espacial menor que lineal.

Se define que una MT M con *una cinta de entrada de sólo lectura y varias cintas de trabajo*, trabaja en espacio S(n) si y sólo si para toda entrada w, tal que |w| = n, M utiliza a lo sumo S(n) celdas en toda cinta de trabajo, en su única computación si es determinística o en cada una de sus computaciones si es no determinística.

De modo similar se define una MT que trabaja en espacio O(S(n)). Cuando S(n) ≥ n se puede utilizar directamente una MT con una cinta de entrada común. Un problema (o lenguaje) pertenece a la clase DSPACE(S(n)), si y sólo si existe una MTD con una cinta de entrada de sólo lectura y varias cintas de trabajo que lo resuelve (o reconoce) en espacio O(S(n)). La misma definición vale para la clase NSPACE(S(n)) considerando las MTN.

La jerarquía espacial está incluida, al igual que la temporal, en la clase R. Si bien en las definiciones anteriores no se explicita que las MT se detienen siempre, se prueba que si existe una MT que trabaja en espacio S(n), entonces existe una MT equivalente que trabaja en el mismo espacio y se detiene siempre. La prueba se basa en que en espacio acotado sólo puede haber un número finito de configuraciones distintas de una MT a partir de una entrada.

**Clase 11**

El estado final de una computación π(S, σ) se denota con val(π(S, σ)). Si π(S, σ) es infinita, se escribe val(π(S, σ)) = ⊥, y se define que val(π(S ; S’, σ)) = ⊥ para todo programa S’ (es decir que la no terminación se propaga). Al símbolo ⊥ se lo conoce como estado indefinido.

**Lenguaje de especificación Assn**

Assn es un lenguaje de primer orden, interpretado sobre el dominio de los números enteros. La interpretación es la estándar, es decir la de los números enteros como los conocemos, considerando sus operaciones y relaciones habituales. Las aserciones tienen la forma:

p :: true | false | e1 = e2 | e1 < e2 | e1 > e2 | … | ¬p | p1 ∨ p2 | p1 ∧ p2 | … | ∃x: p | ∀x: p

**Semántica de las aserciones**: Se utilizan funciones semánticas S(p), que se definen inductivamente en base a la sintaxis de p. Dado un estado σ, S(p)(σ) = verdadero si y sólo si σ satisface p, o en otras palabras, si y sólo si la aserción p es verdadera cuando se evalúa en el estado σ. Como en el caso de las expresiones, podemos simplificar la notación porque trabajamos con interpretaciones fijas. En lugar de S(p)(σ) = verdadero utilizaremos σ |= p:

1. Para toda expresión booleana B, σ |= B ↔ σ(B) = verdadero.
2. σ |= ¬p ↔ ¬ σ |= p. Abreviamos ¬ σ |= p con σ |≠ p. De manera similar se define para la disyunción, la conjunción, etc.
3. σ |= ∃x: p ↔ σ[x | n] |= p para algún número entero n.
4. σ |= ∀x: p ↔ σ[x | n] |= p para todo número entero n.

Además de la visión sintáctica de una aserción, existe una visión semántica: el conjunto de estados que denota, o en otras palabras, todos los estados que la satisfacen. Así, la aserción true denota el conjunto de todos los estados, es decir Ʃ, y la aserción false denota el conjunto vacío de estados, es decir ∅. Expresado de otra manera:

* σ |= true, para todo estado σ.
* σ |≠ false, para todo estado σ.

Las siguientes definiciones de conjuntos de variables facilitarán la notación en las próximas clases:

* var(S): el conjunto de las variables de un programa S.
* change(S): el conjunto de las variables modificables de un programa S, es decir, las variables que aparecen en la parte izquierda de las asignaciones de S.
* var(e): el conjunto de las variables de una expresión e.
* free(p): el conjunto de las variables libres de una aserción p, es decir, las variables de p no alcanzadas por ningún cuantificador.

Una especificación Φ de un programa S de PLW se va a expresar como un par de aserciones (p, q), correspondientes a la entrada y la salida de S, respectivamente. Dada Φ = (p, q), la aserción p se denomina precondición y denota el conjunto de estados iniciales de S, y la aserción q se denomina postcondición y denota el conjunto de estados finales de S.

Ejemplo:

Φ = (x = X, x = 2X)

Es satisfecha por cualquier programa que duplica su entrada x.

Φ = (x = X, x = X + 1)

Es satisfecha por todo programa que incrementa en uno su entrada x.

**Definición 11.5. Correctitud parcial y total de un programa**

Las dos propiedades básicas que debe cumplir un programa secuencial para ser correcto, **correctitud parcial** y **terminación**, que en conjunto conforman su **correctitud total** (en el caso de los programas concurrentes se tienen en cuenta más propiedades, como la ausencia de deadlock, la exclusión mutua y la ausencia de inanición).

Un programa S es **parcialmente correcto** con respecto a una especificación Φ = (p, q), si y sólo si para todo estado σ se cumple

(σ |= p ∧ val(π(S, σ)) ≠ ⊥) → val(π(S, σ)) |= q

En palabras, S es parcialmente correcto con respecto a Φ = (p, q), o directamente (p, q), si y sólo si a partir de cualquier estado σ que satisface la precondición p, si *S termina* lo hace en un estado σ’ que satisface la postcondición q. De esta manera, la correctitud parcial de S con respecto a (p, q) no se viola si existe un estado σ que satisface p a partir del cual S no termina. Justamente, el término parcialmente correcto se debe a que la función semántica M(S) puede asignar a determinados estados iniciales propios el estado indefinido ⊥. La correctitud parcial tampoco se viola cuando se consideran estados que no satisfacen p, independientemente de lo que suceda al final de las computaciones correspondientes.

La expresión |= {p} S {q} denota que S es parcialmente correcto con respecto a (p, q). La fórmula de correctitud {p} S {q} se conoce también como terna de Hoare de correctitud parcial.

Formalmente, se define que un programa S es totalmente correcto con respecto a una especificación (p, q), si y sólo si para todo estado σ se cumple:

σ |= p → (val(π(S, σ)) ≠ ⊥ ∧ val(π(S, σ)) |= q)

Es decir, S es totalmente correcto con respecto a (p, q), si y sólo si a partir de cualquier estado σ que satisface la precondición p, S termina en un estado σ’ que satisface la postcondición q (ahora S es totalmente correcto con respecto a (p, q) porque M(S) asigna a todo estado inicial propio un estado propio). La correctitud total no se viola cuando se consideran estados que no satisfacen p, independientemente de lo que suceda al final de las computaciones correspondientes. La expresión |= 〈p〉 S 〈q〉 denota que S es totalmente correcto con respecto a (p, q). La fórmula de correctitud 〈p〉 S 〈q〉 se conoce también como terna de Hoare de correctitud total.

**Ejemplo 11.3. Fórmulas de correctitud con las aserciones true y false**

La fórmula de correctitud parcial |= {true} S {true} es verdadera cualquiera sea el programa S. En palabras, a partir de un estado σ, todo programa S, si termina, lo hace naturalmente en algún estado σ’. Dados S y σ, debe cumplirse entonces que (σ |= true ∧ val(π(S, σ)) ≠ ⊥) → val(π(S, σ)) |= true. La prueba es la siguiente:

* Se cumple σ |= true.
* Si val(π(S, σ)) = ⊥, entonces la implicación vale trivialmente.
* Si val(π(S, σ)) ≠ ⊥, entonces val(π(S, σ)) = σ’ |= true.

En cambio, la fórmula de correctitud total |= 〈true〉 S 〈true〉 es verdadera sólo en los casos en que S termina cualquiera sea el estado inicial. Y por el contrario, la fórmula de correctitud parcial |= {true} S {false} es verdadera sólo en los casos en que S no termina a partir de ningún estado.

El Lema de Separación, que se formula a continuación, sustenta el modo en que vamos a probar los programas:

Para todo S, p, q: |= 〈p〉 S 〈q〉 ↔ (|= {p} S {q} ∧ |= 〈p〉 S 〈true〉)

Es decir, para probar la fórmula 〈p〉 S 〈q〉, que denota la correctitud total de S con respecto a (p, q), probaremos {p} S {q}, que denota la correctitud parcial de S con respecto a (p, q), y 〈p〉 S 〈true〉, que denota la terminación de S con respecto a la precondición p.

**Definición 11.6. Sensatez y completitud de un método de verificación**

H y H\* son sensatos, propiedad indispensable relacionada con la correctitud de los métodos, que asegura que sólo se prueban fórmulas de correctitud verdaderas. ( *sensato* = de premisas verdaderas llego a resultados verdadero ) H y H\* son completos, propiedad deseable relacionada con el alcance de los métodos, que asegura que se prueban todas las fórmulas de correctitud verdaderas. ( *completo* = con las def que dispone sirve para demostrar cualquier *formula* ).

Considerando genéricamente un método D de verificación de correctitud parcial de programas, se define: D es sensato, si y sólo si para todo programa S y toda especificación (p, q), cumple

|– D {p} S {q} → |= {p} S {q}

D es completo, si y sólo si para todo programa S y toda especificación (p, q), cumple

|= {p} S {q} → |– D {p} S {q}

De manera similar se define la sensatez y la completitud de un método D\* de verificación de terminación de programas.

Como no se definieron fórmulas de correctitud negadas, no existe la posibilidad de que H y H\* sean inconsistentes, es decir que permitan probar fórmulas contradictorias.

**Clase 12 - Verificación de la correctitud parcial**

El método de verificación H, lo usamos probar la correctitud parcial de los programas PLW. Dada una fórmula {p} S {q}, siendo S un programa de PLW y (p, q) una especificación de S constituida por un par de aserciones de Assn, con H se puede encarar el desarrollo de una prueba de {p} S {q}, lo que se denota con la expresión |– H {p} S {q}. Como H es sensato, si se cumple |– H {p} S {q} entonces también se cumple |= {p} S {q}, es decir que con una prueba sintáctica se prueba en realidad que S es parcialmente correcto con respecto a (p, q): dado cualquier estado σ que satisface la precondición p, si S termina a partir de σ entonces lo hace en un estado σ’ que satisface la postcondición q. En cambio, nada se puede decir acerca de la terminación de S.

**Definición 12.1. Axiomas y reglas del método H**

**Axioma del skip (SKIP).**

{p} skip {p}

**Axioma de la asignación (ASI)**

{p[x | e]} x := e {p}

**Regla de la secuencia (SEC)**

{p} S1 {r} , {r} S2 {q}

───────────────────────

{p} S1 ; S2 {q}

**Regla del condicional (COND)**

{ p ∧ B} S1 {q} , {p ∧ ¬B} S2 {q}

─────────────────────────────────

{p} if B then S1 else S2 fi {q}

**Regla de la repetición (REP)**

{p ∧ B} S {p}

────────────────────────────

{p} while B do S od {p ∧ ¬B}

**Regla de consecuencia (CONS)**

p → p1 , {p1} S {q1} , q1 → q

──────────────────────────────

{p} S {q}

**clase 13 Verificación de la terminación**

El método H\* permite probar la terminación de los programas PLW. Una prueba en H\* de la terminación de un programa S a partir de una precondición p, se denota con la expresión |– H\* 〈p〉 S 〈true〉. Por la sensatez del método, |– H\* 〈p〉 S 〈true〉implica |= 〈p〉 S 〈true〉, es decir que a partir de todo estado σ que satisface p, S termina.

Los métodos H y H\* difieren sólo en la regla de la repetición REP, porque el while es la única instrucción que puede provocar no terminación. Como convención de notación, a los nombres de los axiomas y reglas de H\* se les agrega al final el símbolo \*, y las pre y postcondiciones se delimitan con 〈 〉.

**Definición 13.1. Axiomas y reglas del método H\***

Los axiomas SKIP\* y ASI\*, y las reglas SEC\*, COND\* y CONS\*, son los mismos que los del método H (ver Definición 12.1). Las pre y postcondiciones se delimitan con 〈 〉. Por ejemplo, el axioma SKIP\* es 〈p〉 skip 〈p〉. La regla de la repetición REP\* es la siguiente:

**Regla REP\***

p(n + 1) → B , 〈p(n + 1)〉 S 〈p(n)〉 , p(0) → ¬B

────────────────────────────────────────────────────

〈∃n: p(n)〉 while B do S od 〈p(0)〉

**Regla REP\*\***

〈p ∧ B〉 S 〈p〉 , 〈p ∧ B ∧ t = Z〉 S 〈t < Z〉 , p → t ≥ 0

──────────────────────────────────────────────────────────────

〈p〉 while B do S od 〈p ∧ ¬B〉