TEORÍA DE LA COMPUTACIÓN Y VERIFICACIÓN DE PROGRAMAS

FACULTAD DE INFORMÁTICA / UNIVERSIDAD NACIONAL DE LA PLATA

TEORÍA DE LA COMPUTACIÓN Y VERIFICACIÓN DE PROGRAMAS

RICARDO ROSENFELD / JERÓNIMO IRAZÁBAL

Mc Graw Hill Educación

edulp Editorial de la Universidad de La Plata

Rosenfeld, Ricardo
Teoría de la computación y verificación de programas / Ricardo Rosenfeld y Jerónimo Irazábal. - 1a ed. - La Plata: Universidad Nacional de La Plata, 2010.
420 p. ; 22x16 cm.

ISBN 978-950-34-0633-5

1. Informática. 2. Programación. 3. Enseñanza Universitaria. I. Irazábal, Jerónimo II. Título
CDD 005.3

Fecha de catalogación: 25/02/2010

TEORÍA DE LA COMPUTACIÓN Y VERIFICACIÓN DE PROGRAMAS

RICARDO ROSENFELD | JERÓNIMO IRAZÁBAL

Diagramación: Andrea López Osornio

Editorial de la Universidad Nacional de La Plata
Calle 47 Nº 380 - La Plata (1900) - Buenos Aires - Argentina
Tel/Fax: 54-221-4273992
e-mail: editorial_unlp@yahoo.com.ar
www.unlp.edu.ar/editorial

La EDULP integra la Red de Editoriales Universitarias (REUN)

1º edición - 2010
ISBN Nº 978-950-34-0633-5
Queda hecho el depósito que marca la ley 11.723

Impreso en Argentina

ÍNDICE

PREFACIO

Este libro trata, de una manera introductoria, sobre la computabilidad y la complejidad de los problemas, y la verificación de la correctitud de los programas de computadora. Su contenido se basa en el de las asignaturas Teoría de la Computación y Verificación de Programas y Teoría de la Computación y Verificación de Programas Avanzada, que dicto desde hace tiempo para cuarto y quinto año de la Licenciatura en Informática de la Universidad Nacional de La Plata.

Por lo tanto, este material está dirigido a los alumnos de los últimos años de las carreras de computación, y a graduados que deseen introducirse en las áreas mencionadas. Se asume que el lector tiene cierta madurez matemática, y que ya ha adquirido conocimientos sobre algorítmica y estructuras de datos, conceptos y paradigmas de los lenguajes de programación, y desarrollo de programas.

Hasta hace poco tiempo he dictado por separado Teoría de la Computación en una asignatura y Verificación de Programas en otra. Posteriormente fueron unificadas sus partes básicas en una primera asignatura, y el resto de los temas se agruparon en una asignatura avanzada. Una manera de leer el libro, que se explica más adelante, se basa en este esquema de dictado.

El libro es un viaje imaginario que arranca en los confines del universo de problemas y se dirige hacia su interior. Al comienzo se reconocen y atraviesan las fronteras de la computabilidad, primero, y de la decidibilidad después, recorriéndose de esta manera los problemas no computables, los problemas semidecidibles y los problemas decidibles. Estos contenidos conforman la Parte I, Computabilidad y Decidibilidad. Se introducen las máquinas de Turing como modelo computacional, se definen los problemas (y lenguajes asociados) tratables por las máquinas de Turing, es decir los problemas computables, y dentro de este conjunto se identifican los problemas semidecidibles, a los que en algunas instancias las máquinas de Turing no logran resolver. Se describe y aplica, además, la técnica de reducción de problemas, muy útil para clasificar a los problemas de acuerdo a la taxonomía anterior.

El segundo tramo del viaje imaginario explora específicamente los problemas decidibles, con el objeto de analizar su complejidad computacional, en términos del tiempo (pasos) y el espacio (celdas de memoria) consumidos por las máquinas de Turing que los resuelven. Se reconoce y atraviesa la frontera entre los problemas intratables y los problemas tratables, entendiendo por tratable a un problema que se puede resolver por una máquina de Turing que no consume más de un determinado tiempo y un determinado espacio, de acuerdo a una convención que se formaliza y justifica adecuadamente. Estos contenidos conforman la Parte II, Complejidad Computacional. Se presentan jerarquías temporales y espaciales de clases de problemas según su complejidad computacional, y las jerarquías se relacionan entre sí para obtener una única clasificación temporal-espacial.

En el último recorrido del viaje mantenemos nuestro rumbo hacia los problemas decidibles, pero ahora con otro propósito, el de plantear una metodología para verificar la correctitud de los programas que se construyen para resolver los problemas. Estos contenidos conforman la Parte III, Verificación de Programas. Se describe un método axiomático para probar distintas propiedades de los programas, empezando por los programas secuenciales determinísticos, pasando por los programas secuenciales no determinísticos, y culminando en los programas concurrentes.

El desarrollo de los temas combina rigor matemático con informalidad para introducir los conceptos básicos. Se presentan ejemplos que acompañan las definiciones y los teoremas. De todos modos, el libro no pretende compendiar resultados y ejemplos sobre computabilidad, complejidad y verificación de programas, sino que el foco está en plantear, en base a casos representativos, las ideas fundamentales de las distintas áreas. En todos los casos se ha optado por emplear la manera más intuitiva y difundida en la literatura para presentar los temas. Por ejemplo, se emplean las máquinas de Turing como modelo computacional, se consideran los recursos de tiempo y espacio de las máquinas de Turing para medir la complejidad de un problema, y la semántica de los distintos lenguajes de programación utilizados en la parte del libro dedicada a la verificación de programas se describe operacionalmente.

Cada una de las tres partes en que se estructura el libro es un pequeño libro en sí mismo, con su introducción, capítulos con los contenidos del área correspondiente, apéndice, referencias bibliográficas y ejercicios. Previamente se introducen algunas definiciones generales que se utilizan en todo el libro. En el apéndice se presentan temas que

no se consideran centrales con respecto al grado introductorio del material. Además de los ejercicios al final de cada parte, se incluyen otros tan o más importantes a lo largo de los capítulos, para involucrar al lector en la terminación de la prueba de un teorema, o para reforzar su entendimiento acerca de una definición.

Una de las formas recomendadas para leer este libro es naturalmente la secuencial, desde el comienzo hasta el final. Pero también existen otras posibilidades. Por ejemplo, podrían dejarse los tres apéndices para el final. También podría considerarse primero la parte de verificación de programas, y luego las de computabilidad y complejidad en este orden. Una cuarta posibilidad coincide con la actual modalidad de dictado de las dos asignaturas referidas, en la Facultad de Informática de la Universidad Nacional de La Plata: (a) la primera parte completa (computabilidad y decidibilidad), (b) los dos primeros capítulos de la segunda parte (complejidad computacional temporal), (c) los dos primeros capítulos de la tercera parte (verificación de programas secuenciales determinísticos) –esto constituye actualmente la asignatura básica–, (d) los dos últimos capítulos de la segunda parte (complejidad computacional espacial) más el apéndice, y (e) los dos últimos capítulos de la tercera parte (verificación de programas secuenciales no determinísticos y concurrentes) más el apéndice –esto constituye actualmente la asignatura avanzada–.

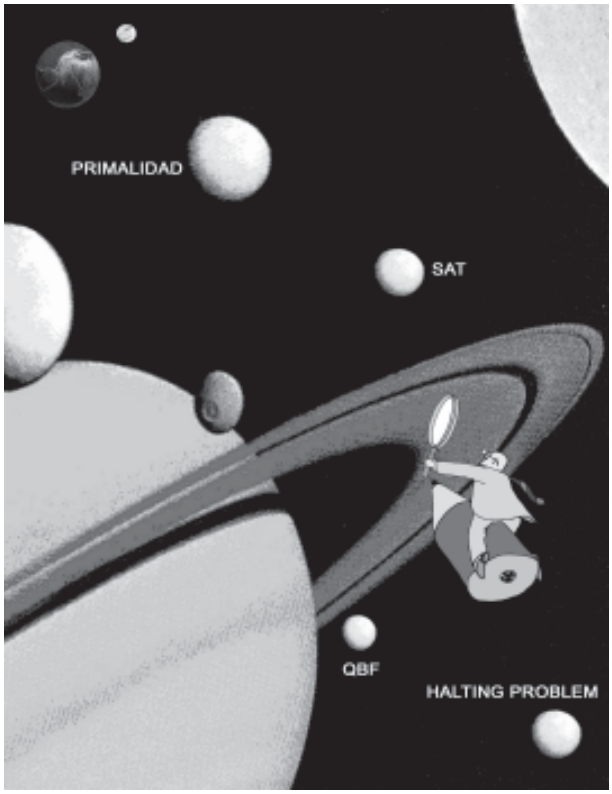
Quiero agradecer especialmente a Jerónimo Irazábal, colaborador desde hace varios años en las asignaturas que dicto. No sólo llevó a cabo la investigación bibliográfica que acompañó la confección de este libro, sino que también elaboró gran parte de los ejercicios, diseñó los gráficos, y fundamentalmente discutió y revisó conmigo la redacción del material, párrafo a párrafo. Por eso es que Jerónimo Irazábal es co-autor de Teoría de la Computación y Verificación de Programas.

Agradezco a las autoridades de la Facultad de Informática de la Universidad Nacional de La Plata, por haberme dado la posibilidad de esta publicación. Agradezco a mi familia, que durante meses toleró mis ausencias y me alentó permanentemente. Agradezco a los anteriores colaboradores en las asignaturas mencionadas, en especial a Roberto Figueroa y Pablo Mocciola, porque en alguna medida ellos también son co-autores de este libro. Agradezco a Victoria Meléndez, por haber realizado la ilustración principal del libro. Quiero cerrar la lista de agradecimienos refiriéndome particularmente a mis alumnos, quienes a lo largo de estos años también fueron modelando la presentación de estos temas, con sus opiniones, preguntas y críticas siempre constructivas; este material es por ellos y para ellos.

Mi mayor deseo es que este libro sirva como guía de estudio, que contribuya a la formación básica de los alumnos de las carreras de computación, y que estimule la profundización en algunos tópicos de la teoría de la computación (por ejemplo en las distintas variantes de las máquinas de Turing presentadas en el Apéndice 2), y de la teoría de correctitud de programas (por ejemplo en la verificación de los programas concurrentes, en la verificación de los programas reactivos empleando lógica temporal que se presenta en el Apéndice 3, y en el desarrollo sistemático de programas). También espero que los temas entretengan al lector tanto como a mí. Están todos invitados a sugerir mejoras a este material. ¡Gracias desde ya!

RICARDO ROSENFELD

La Plata, 21 de julio de 2009



DEFINICIONES PRELIMINARES

Las siguientes definiciones y notaciones se utilizan en el libro. Algunas definiciones se reformulan después. Si bien asumimos que en general, son conocidas por el lector, recomendamos revisarlas antes de iniciar la lectura del libro.

1. Conjuntos

Un conjunto es una colección de elementos. La expresión A = {a, b, c} denota que el conjunto A está formado por los elementos a, b y c. En este caso se cumple que a pertenece a A (a ε A) y que d no pertenece a A (d $\notin$ A). Los conjuntos A y B son iguales (A = B) si tienen los mismos elementos. A es el conjunto vacío (A = $\varnothing$) si no tiene elementos. A es finito o infinito si tiene una cantidad finita o infinita de elementos, respectivamente.

La expresión A = {a | a ε B y a satisface P} denota que el conjunto A tiene todos los elementos del conjunto B que satisfacen la propiedad P. Por ejemplo, si N es el conjunto de los números naturales, A = {a | a ε N y a es par} denota el conjunto de los números naturales pares. También se puede escribir A = {a | a ε N $\wedge$ a es par}, donde «$\wedge$» es un conectivo lógico que denota «y». Otros conectivos lógicos que se utilizan son «$\vee$» para «o», «$\neg$» para «no», «$\rightarrow$» para «entonces», y «$\leftrightarrow$» para «si y solo si», que se abrevia «sii». También se utilizan los cuantificadores universal y existencial, que respectivamente son «$\forall$» ($\forall$a se lee «para todo a») y «$\exists$» ($\exists$a se lee «existe a»).

Un conjunto A es un subconjunto de un conjunto B (A $\subseteq$ B) si todos los elementos de A pertenecen a B. A es un subconjunto propio de B (A $\subset$ B) si A $\subseteq$ B y A $\neq$ B. También se dice en este caso que B incluye estrictamente a A. Se cumple que A = B $\leftrightarrow$ (A $\subseteq$ B $\wedge$ B $\subseteq$ A).

La unión de A y B (A $\cup$ B) es el conjunto formado por los elementos que están en A o en B. La intersección de A y B (A $\cap$ B) es el conjunto formado por los elementos que están en A y en B. Los conjuntos A y B son disjuntos si A $\cap$ B = $\varnothing$. La diferencia de A y B (A – B) es el conjunto formado por los elementos de A que no están en B.

La cardinalidad de un conjunto A es la cantidad de sus elementos, y se denota con |A|. El cardinal del conjunto infinito N se denota con $\aleph_0$, y el primer ordinal infinito con ω. Se va a recurrir a los ordinales

infinitos cuando se trate la verificación de los programas no determinísticos.

El conjunto de partes de A, denotado con la expresión 2^A (también se utiliza la expresión $\mathcal{P}(A)$), es el conjunto de todos los subconjuntos de A, incluyendo el propio A y el conjunto vacío. Por ejemplo, {∅, {a}, {b}, {a, b}} es el conjunto de partes de {a, b}. Si la cardinalidad de A es n, entonces la cardinalidad de 2^A es 2^n. Se prueba, cualquiera sea el caso, que la cardinalidad del conjunto de partes de un conjunto es mayor que la cardinalidad del conjunto.

Una partición de un conjunto $A \neq \emptyset$, es un subconjunto B de 2^A tal que (a) todo conjunto de B no es vacío, (b) los conjuntos de B son dos a dos disjuntos, y (c) la unión de los conjuntos de B es igual al conjunto A. Por ejemplo, {{a}, {b, c}} es una partición de {a, b, c}.

2. Relaciones y funciones

La expresión <a, b> denota el par ordenado (o simplemente el par) formado por los elementos a y b. También se utiliza la notación (a, b). El par <a, b> es distinto del conjunto binario {a, b}, porque en el primer caso se considera el orden de los componentes y en el segundo no.

El producto cartesiano de A y B (A x B) es el conjunto de todos los pares <a, b> tales que $a \varepsilon A$ y $b \varepsilon B$. Cualquier subconjunto R de A x B es una relación binaria entre A y B. El dominio de R es el conjunto de todos los elementos a tales que $<a, b> \varepsilon R$ para algún b, y el codominio o rango de R es el conjunto de todos los elementos b tales que $<a, b> \varepsilon R$ para algún a. A toda relación binaria R se le asocia el predicado R(a, b), que tiene el valor verdadero sii $<a, b> \varepsilon R$. Con la expresión R(a, b) = tt (o true), se denota que R(a, b) es verdadero (también directamente se puede expresar R(a, b)), mientras que la expresión R(a, b) = ff (o false), denota que R(a, b) es falso (también directamente se puede expresar ¬R(a, b)).

El concepto de par <a, b> se puede extender al de n-tupla $<a_1, a_2, \ldots, a_n>$. El producto cartesiano de $A_1, A_2, \ldots, A_n$ es el conjunto $A_1 \times A_2 \times \ldots \times A_n = \{<a_1, a_2, \ldots, a_n> \mid a_1 \varepsilon A_1 \wedge a_2 \varepsilon A_2 \wedge \ldots \wedge a_n \varepsilon A_n\}$. Cualquier subconjunto R de $A_1 \times A_2 \times \ldots \times A_n$ es una relación n-aria entre $A_1, A_2, \ldots, A_n$. El producto cartesiano de n conjuntos A se denota con A^n.

Dado un conjunto A, un número natural $n > 0$ y una relación $R \subseteq A^{n+1}$, un conjunto $B \subseteq A$ es un conjunto cerrado con respecto a la relación R si para toda (n+1)-tupla $<b_1, b_2, \ldots, b_{n+1}> \varepsilon R$, se cumple que $(b_1 \varepsilon B \wedge b_2 \varepsilon B \wedge \ldots \wedge b_n \varepsilon B) \rightarrow b_{n+1} \varepsilon B$. Por ejemplo, dada $R_{sum} \subseteq N^3$, que es la relación de suma de dos números naturales, se cumple que N es cerrado con respecto a R_{sum}.

Una relación binaria R entre A y A también se refiere como una relación binaria en A, y se denota con (A, R). Se dice que R es una relación reflexiva si para todo elemento a del dominio de R, se cumple R(a, a), y es irreflexiva si para todo elemento a del dominio de R, se cumple $\neg$R(a, a). R es simétrica si para todo par de elementos a y b, se cumple $R(a, b) \rightarrow R(b, a)$, y es antisimétrica si para todo par de elementos a y b, se cumple $(R(a, b) \wedge R(b, a)) \rightarrow a = b$. La relación R es transitiva si para toda terna de elementos a, b y c, se cumple $(R(a, b) \wedge R(b, c)) \rightarrow R(a, c)$.

Una relación binaria R en A que es reflexiva, antisimétrica y transitiva, se denomina relación de orden parcial. Si en cambio R es irreflexiva y transitiva (y por lo tanto también antisimétrica), se denomina relación de orden parcial estricta (también se dice que R es un orden parcial estricto, y que A es un conjunto parcialmente ordenado estricto). En este último caso se utiliza, en general, la notación (A, <). Por ejemplo, se cumple que (N, <) es un conjunto parcialmente ordenado estricto, siendo < la relación habitual entre números.

Dado (A, <), un elemento $a \varepsilon A$ es un minimal de A si no existe ningún elemento $b \varepsilon A$ tal que $b < a$. Si no existe ninguna secuencia infinita $\{a_n\}_{n \geq 0}$ tal que $a_{n+1} < a_n$, entonces A es un conjunto parcialmente ordenado estricto bien fundado, o directamente un conjunto parcialmente ordenado bien fundado (por ejemplo, otra vez se cumple en el caso de N con respecto a la relación < habitual).

Una función f del conjunto A al conjunto B ($f : A \rightarrow B$), es una relación binaria entre A y B que tiene a lo sumo un par <a, b> para todo $a \varepsilon A$. La expresión f(a) = b denota que $<a, b> \varepsilon f$, siendo a el argumento y b el valor de f. Una función $f : A \rightarrow B$ es total si su dominio coincide con A, y es parcial en caso contrario; es inyectiva si para todo par de elementos distintos a y a' de A, se cumple que $f(a) \neq f(a')$; es suryectiva si su codominio coincide con B; y es biyectiva si es inyectiva y suryectiva (también se dice en este caso que existe una biyección entre A y B).

Una función inyectiva $f : A \rightarrow B$ admite la función inversa $f^{-1} : B \rightarrow A$, tal que $f(a) = b \leftrightarrow f^{-1}(b) = a$. Por ejemplo, $f : N \rightarrow N$ tal que f(n) = n admite la función inversa $f^{-1} = f$, y la función $f : N \times N \rightarrow N$ tal que f(a, b) = a + b no admite función inversa, porque no es inyectiva.

Dadas las funciones $f : N \to N$ y $g : N \to N$, la expresión $f = O(g)$ denota que existe una constante c tal que para todo número natural n se cumple que $f(n) \leq c.g(n)$. Se dice que la función f es del orden de la función g. Por ejemplo, es el caso de $f(n) = n^2$ y $g(n) = 2^n$.

Un conjunto A es contable (o numerable, o enumerable) si es finito, o si es infinito y existe una biyección entre A y N. La idea es que los elementos de A se puedan listar en un determinado orden. Por ejemplo, se prueba que el conjunto de los números enteros es contable, y que el conjunto de los números reales no es contable. Los conjuntos numerables de strings serán un tema central en la parte del libro dedicada a la computabilidad y decidibilidad.

3. Lenguajes

Un alfabeto es un conjunto finito no vacío de símbolos $\Sigma = \{x_1, x_2, \ldots, x_k\}$. Un string (o cadena de símbolos, o palabra) es una tupla finita de símbolos de Σ. El string vacío se denota con λ. El conjunto infinito de todos los strings que se pueden formar con los símbolos de Σ se denota con Σ^*, mientras que $\Sigma^+ = \Sigma^* - \{\lambda\}$. La expresión $|w|$ denota el tamaño o la cantidad de símbolos del string w.

El orden canónico (o lexical canónico, o lexicográfico) sobre Σ^* se define de la siguiente manera: un string precede a otro de tamaño mayor, y sobre los strings de igual tamaño se establece el orden alfabético. Por ejemplo, si $\Sigma = \{x, y, z\}$, los primeros strings de Σ^* según el orden canónico son: λ, x, y, z, xx, xy, xz, yx, yy, yz, zx, zy, zz, xxx, xxy, xxz, xyx, …

Todo subconjunto de Σ^* es un lenguaje (a partir de Σ). Los lenguajes incluidos en Σ^*, cuando $|\Sigma| = 1$, se denominan lenguajes tally. El complemento de un lenguaje L se denota con L^C y se define como $L^C = \{w \mid w \varepsilon \Sigma^* \wedge w \notin L\}$. La concatenación o producto de dos lenguajes se denota con $L_1 \bullet L_2$ y se define como $L_1 \bullet L_2 = \{w \mid \exists w_1, w_2 : w_1 \varepsilon L_1, w_2 \varepsilon L_2, w = w_1 w_2\}$. En particular, si L^n denota el producto del lenguaje L por sí mismo n veces, entonces se define como clausura de L al lenguaje $L^* = \cup_{n \geq 0} L^n$, siendo por convención $L^0 = \{\lambda\}$.

Con $L_{\leq n}$ se denota el subconjunto de L con strings de tamaño a lo sumo n. La densidad de un lenguaje L se determina por la razón de crecimiento de la función censo c_L de L, que se define como $c_L : n \to |L_{\leq n}|$. Los lenguajes finitos son los de menor densidad, porque $c_L(n)$ es constante cuando n es grande. Si $c_L(n)$ está acotado por un polinomio

en n, se dice que el lenguaje L es esparso. En particular, los lenguajes tally son esparsos. Los lenguajes más densos son aquéllos cuya función censo crece exponencialmente con n. Por ejemplo, si $\Sigma = \{0, 1\}$ y <n> denota la representación binaria del número natural n, entonces L = {<n> | n es una potencia de 2} es un lenguaje esparso, y L = {<n> | n es par} no lo es. La densidad de un lenguaje es un propiedad útil para caracterizarlo en el marco de la complejidad computacional, como se verá más adelante.

Una manera de representar lenguajes es por medio de gramáticas. Por ejemplo, utilizando la notación Backus-Naur que se verá en la tercera parte del libro, un lenguaje de expresiones aritméticas formado por los símbolos 0, 1, v, +, –, (,), se puede representar por la gramática:
$e ::= 0 \mid 1 \mid v \mid (e_1 + e_2) \mid (e_1 - e_2)$.

4. Grafos

Un grafo G es un par (V, E), tal que V es un conjunto de vértices y E es un conjunto de arcos. Un arco está determinado por dos vértices, que se dicen adyacentes. Cuando se establece un orden entre los vértices de un arco, el grafo es orientado o dirigido. En algunos casos se asocia a cada arco un valor o peso, en cuyo caso se dice que el grafo está ponderado o que es un grafo con pesos. La cantidad de vértices adyacentes de un vértice v es el grado de v. El grado de un grafo es el máximo grado considerando todos sus vértices.

Un grafo es completo (o es un clique) si todo par de vértices determina un arco. Un grafo G' = (V', E') es un subgrafo de G = (V, E), si $V' \subseteq V$ y $E' \subseteq E$. Dado un grafo, un camino de longitud k desde el vértice v_1 hasta el vértice v_2 es una secuencia de k arcos $(v_1,v_{i1}),(v_{i1},v_{i2}),\ldots,(v_{ik-1},v_2)$. En particular, si $v_1 = v_2$, dicha secuencia se denomina circuito o ciclo. Un grafo es conexo si entre todo par de vértices existe un camino.

Un árbol es un grafo conexo que no tiene circuitos. En un árbol, los vértices se conocen también como nodos. En los árboles orientados se distinguen distintos tipos de nodos, como se explica a continuación. El nodo raíz es el primer nodo del árbol. Un ancestro de un nodo v, es un nodo v' intermedio en el camino del nodo raíz al nodo v; en este caso, se dice que v es descendiente de v'. Un ancestro inmediato de un nodo se denomina nodo padre, y un descendiente inmediato se denomina nodo hijo. Los nodos sin hijos se denominan hojas. La longitud entre el

nodo raíz y un nodo v es la profundidad de v. La profundidad de un árbol orientado es la máxima profundidad, considerando todos sus nodos.

5. Técnicas para probar teoremas

Algunas técnicas para probar teoremas que se utilizarán en el libro son las siguientes:

1) Inducción:

Para probar que una propiedad P se cumple en todos los números naturales, se puede hacer:

- Base de la inducción: probar que la propiedad P vale en n = 0.
- Paso inductivo: asumir que la propiedad P vale en n = k (hipótesis inductiva), y probar que P vale en n = k + 1.

También se puede utilizar la inducción para desarrollar definiciones basadas en los números naturales.

Un caso más general de inducción es la inducción estructural, que permite más de un caso para la base de la inducción y para el paso inductivo. Ahora se puede tratar con cualquier conjunto parcialmente ordenado bien fundado. La prueba consiste en demostrar que una propiedad se cumple en todos los minimales, y que si la propiedad se cumple en todas las subestructuras de una cierta estructura, entonces también se cumple en la estructura. Por ejemplo, este principio se puede utilizar para definir el lenguaje presentado antes de las expresiones aritméticas compuestas por las constantes 0 y 1, la variable v, y los operadores + y – . Se puede hacer:

- Base de la inducción: 0, 1 y v son expresiones aritméticas.
- Paso inductivo: si e_1 y e_2 son expresiones aritméticas, entonces (e_1 + e_2) y (e_1 – e_2) también lo son.

2) *Breadth-first search* (búsqueda primero a lo ancho):

Un ejemplo muy representativo de la aplicación de esta técnica es el siguiente. Dado un conjunto contable de conjuntos infinitos contables

A_0, A_1, A_2, …, y un elemento a, para chequear que a ε A_i para algún i, se puede hacer:

- Chequear si a es el primer elemento de A_0.
- Chequear si a es el segundo elemento de A_0, o el primer elemento de A_1.
- Chequear si a es el tercer elemento de A_0, o el segundo elemento de A_1, o el primer elemento de A_2.
- Y así sucesivamente.

Notar que si, por ejemplo, x ε A_1, esto no se detectará si se chequea primero en todo A_0, es decir si se chequea a lo largo (técnica de *depth-first search*). La técnica de depth-first search es una alternativa válida cuando se trata con conjuntos finitos.

3) Diagonalización:

Un ejemplo muy representativo de la aplicación de esta técnica es el siguiente. Supóngase que se quiere probar que todo conjunto contable A de subconjuntos de un conjunto infinito contable B es distinto de 2^B. La prueba se basa en la obtención de un subconjunto de B que no está en A:

Llamando A_i a los conjuntos de A, y b_i a los elementos de B, con $i \geq 0$, sea $D = \{b_i \mid b_i \notin A_i\}$. Se cumple que $D \subseteq B$ y que para todo i, $D \neq A_i$, por lo que $A \neq 2^B$. Se cumple $D \subseteq B$ por definición. Y se cumple $D \neq A_i$ para todo i: $D \neq A_0$ porque difieren en el elemento b_0, $D \neq A_1$ porque difieren en el elemento b_1, y así sucesivamente.

Esta técnica de prueba se denomina diagonalización, porque se la puede relacionar con la diagonal de una matriz infinita de unos y ceros. En efecto, notar que si M es una matriz infinita tal que $M_{ik} = 0$ si $b_i \varepsilon A_k$, y $M_{ik} = 1$ si $b_i \notin A_k$, entonces la diagonal de M representa el conjunto D, dado que se cumple $M_{ii} = 1$ sii $b_i \varepsilon D$.

6. Fórmulas booleanas

La sintaxis de una fórmula booleana se define inductivamente de la siguiente manera. Dado un conjunto contable $X = \{x_1, x_2, x_3, ...\}$ de variables booleanas:

- Toda variable booleana $x_i \varepsilon X$ es una fórmula booleana.
- Si E_1 y E_2 son fórmulas booleanas, también lo son $(E_1 \vee E_2)$, $(E_1 \wedge E_2)$ y $\neg E_1$.

Los paréntesis redundantes pueden omitirse. Una fórmula booleana de la forma x_i ó $\neg x_i$ se denomina literal. Si bien las fórmulas booleanas también se conocen como expresiones booleanas, esta segunda denominación se utilizará en general sólo en la tercera parte del libro, en que se va a considerar otra sintaxis.

Desde el punto de vista semántico, una fórmula booleana puede ser verdadera o falsa, lo que depende de los valores de verdad de sus variables booleanas, determinados por una asignación A. Una asignación A satisface una fórmula booleana E, si cuando E se evalúa con A resulta verdadera, lo que se denota con $A \models E$. Se define inductivamente:

- Si $E = x_i$, entonces $A \models E$ si $A(x_i) = tt$.
- Si $E = \neg E_1$, entonces $A \models E$ si $A \not\models E_1$.
- Si $E = (E_1 \vee E_2)$, entonces $A \models E$ si $A \models E_1$ ó $A \models E_2$.
- Si $E = (E_1 \wedge E_2)$, entonces $A \models E$ si $A \models E_1$ y $A \models E_2$.

Una fórmula booleana cuantificada es una fórmula booleana con los cuantificadores universal o existencial. Su sintaxis se define inductivamente de la siguiente manera. Dado un conjunto contable $X = \{x_1, x_2, x_3, ...\}$ de variables booleanas:

- Toda variable booleana $x_i \varepsilon X$ es una fórmula booleana cuantificada. En este caso, x_i es una variable libre. También se dice que la ocurrencia de x_i es libre.
- Si E_1 y E_2 son fórmulas booleanas cuantificadas, también lo son $(E_1 \vee E_2)$, $(E_1 \wedge E_2)$ y $\neg E_1$. En este caso, toda variable x_i es una variable libre o ligada, según sea una variable libre o ligada en E_1 o E_2, respectivamente.
- Si E es una fórmula booleana cuantificada, también lo son $\forall x_i(E)$ y $\exists x_i(E)$. En este caso, toda variable x_i libre en E es una variable ligada, y toda otra variable x_k es una variable libre o ligada, según sea una variable libre o ligada en E, respectivamente.

Desde el punto de vista semántico, una fórmula booleana cuantificada sin variables libres puede ser verdadera o falsa. El valor se determina reemplazando cada subfórmula de la forma $\forall x_i(E)$ por $E_1 \wedge E_2$, y cada subfórmula de la forma $\exists x_i(E)$ por $E_1 \vee E_2$, tal que E_1 y E_2 son como E

pero con todas las ocurrencias de x_i alcanzadas por el cuantificador, reemplazadas en E_1 por el valor tt y en E_2 por el valor ff.

7. Sistemas deductivos

Un sistema deductivo o cálculo D produce fórmulas a partir de un conjunto de axiomas (conjunto inicial de fórmulas) y un conjunto de reglas de inferencia o directamente reglas.

Por medio de las reglas se pueden producir nuevas fórmulas en D. La forma de una regla es:

$$\frac{\vartheta_1, \ldots, \vartheta_k}{\vartheta}$$

donde las fórmulas $\vartheta_1, \ldots, \vartheta_k$ se denominan premisas y la fórmula ϑ se denomina conclusión. En este caso se establece que de las premisas $\vartheta_1, \ldots, \vartheta_k$ se produce o se deduce la conclusión ϑ.

La prueba o demostración de una fórmula ϑ en D es una secuencia finita de fórmulas $\vartheta_1, \ldots, \vartheta_n$, donde $\vartheta = \vartheta_n$ y toda fórmula ϑ_i, con $1 \leq i \leq n$, es un axioma o puede ser obtenida por la aplicación de una o más reglas sobre las fórmulas $\vartheta_1, \ldots, \vartheta_{i-1}$. En particular, ϑ_1 es un axioma, y ϑ_n se denomina teorema. La longitud de la secuencia de fórmulas es la longitud de la prueba.

Dado D y una fórmula ϑ, la expresión $\vdash_D \vartheta$ denota que ϑ es un teorema de D.

PARTE I
COMPUTABILIDAD Y DECIDIBILIDAD

Introducción de la Parte I

El primer tramo de nuestro viaje imaginario arranca en los confines del universo de problemas. Nos vamos a dirigir hacia su interior. Recorreremos problemas no computables, problemas semidecidibles y problemas decidibles. Obtendremos una visión general de este universo. Hacia los problemas decidibles nos iremos acercando en los siguientes tramos del viaje…

El tema de la primera parte del libro es la *computabilidad* y la *decidibilidad*. Esto significa que el concepto fundamental que será tratado es el *algoritmo*.

Un algoritmo es un conjunto finito de instrucciones claramente expresadas, que tiene como propósito solucionar un problema. Con estas características, un algoritmo no sólo puede ser ejecutado por un ser humano sino también por una máquina, dado que con las instrucciones, un estado inicial y un dato de entrada, se cuenta con toda la información necesaria. Es decir, un algoritmo es un procedimiento mecánico. Cuando no se exige que la ejecución termine, en lugar de algoritmo (o *procedimiento efectivo*) se habla solamente de *procedimiento*.

El modelo más popular e intuitivo de máquina para ejecutar algoritmos y procedimientos es la *máquina de Turing*.

Las máquinas de Turing están profundamente relacionadas con el formalismo lógico. A comienzos del siglo XX se planteaba si acaso podía existir un algoritmo general para decidir qué fórmulas son teoremas, en el marco del cálculo de primer orden. La respuesta negativa a este

problema (*Entscheidungsproblem*) la dio A. Turing en 1936; en la prueba introdujo la máquina que lleva su nombre, con el objeto de derivar teoremas a partir de axiomas y reglas. El mismo año A. Church también lo demostraba, utilizando otro formalismo, y ya en 1931 K. Gödel mostraba mediante su famoso teorema de incompletitud que en toda axiomática que describe la aritmética de los números naturales existen proposiciones verdaderas que no pueden probarse.

Las máquinas de Turing se consideran desde entonces una adecuada formalización de la noción de «lo computable». Existen otras formalizaciones equivalentes, como por ejemplo el cálculo Lambda de Church (ya referido), las funciones recursivas parciales de Kleene, los algoritmos de Markov, y los sistemas de Post. Lo interesante es que a tantas décadas de la aparición de estos modelos, la conjetura conocida como *tesis de Church* (o *tesis de Church-Turing*), de que todo lo computable puede ser llevado a cabo por una máquina de Turing, no ha podido ser refutada. Así las cosas, asumiendo que se cumple la tesis de Church, las máquinas de Turing modelizan las computadoras.

La computabilidad y la decidibilidad se desarrollarán en términos de las máquinas de Turing. La computabilidad trata sobre los problemas computables y no computables por una máquina de Turing. Las máquinas de Turing no necesariamente terminan a partir de cualquier dato de entrada. Así surge el concepto de la decidibilidad, que trata sobre los problemas decidibles, para los que existen máquinas de Turing que terminan a partir de cualquier dato de entrada, y los problemas semidecidibles, sobre los que las máquinas de Turing asociadas no terminan en algunos casos.

En el Capítulo 1.1 se introducen las máquinas de Turing. Se definen distintos modelos equivalentes, que después se utilizarán indistintamente para facilitar las demostraciones de los teoremas y la presentación de los ejemplos.

En los Capítulos 1.2 y 1.3, en conjunto, se demuestra la existencia de problemas no computables, semidecidibles y decidibles. Es decir, se trazan formalmente las fronteras, dentro del universo de problemas, de lo computable y lo decidible.

Se utiliza fundamentalmente la visión de máquina de Turing que reconoce un lenguaje, según la cual resolver un problema (en realidad, un problema de decisión) consiste en reconocer un lenguaje. La idea es que los strings (o cadenas) de un lenguaje representan instancias de un problema. Desde este punto de vista, resolver por ejemplo el problema de determinar si un grafo tiene un camino del vértice v_1 al vértice v_m, consiste en construir una máquina de Turing M que, dado un string w que representa un grafo, se detenga y acepte w si el grafo tiene un

camino de v_1 a v_m, y se detenga y rechace w en caso contrario. De esta manera se hablará indistintamente de problemas y lenguajes. Los lenguajes asociados a los problemas semidecidibles se denominan recursivamente numerables, y los que están asociados a los problemas decidibles son los lenguajes recursivos. Se demuestran propiedades de ambas clases de lenguajes.

En el Capítulo 1.4 se introduce la técnica de reducción de problemas. La idea de esta técnica es tan simple como efectiva: para resolver un problema nuevo, en lugar de buscar de cero una solución, se lo reduce a un problema del que ya se conoce su solución. Mediante esta técnica se va a probar que importantes problemas de la ciencia de la computación no son decidibles.

Finalmente, en el Apéndice 1 se describen suscintamente máquinas de Turing restringidas, es decir máquinas de Turing con poder computacional limitado, destinadas a resolver problemas específicos. La clasificación de las mismas se basa en la jerarquía de Chomsky de la teoría de los lenguajes formales. Problemas indecidibles con máquinas de Turing generales, pueden llegar a ser decidibles cuando se trata con máquinas de Turing restringidas. Otros capítulos complementarios del apéndice presentan la descripción de la visión de máquina de Turing como generadora de un lenguaje, la representación alternativa de los lenguajes mediante las gramáticas, y la prueba de equivalencia entre las máquinas de Turing y las máquinas RAM, relación interesante dada la similitud estructural entre estas últimas y las computadoras, lo que refuerza la tesis de Church.

CAPÍTULO 1.1. MÁQUINAS DE TURING

En este capítulo se introducen las máquinas de Turing. Se considera inicialmente sólo la visión de máquina de Turing como reconocedora de un lenguaje.
Las máquinas de Turing fueron presentadas por A. Turing en un artículo de 1936. La idea básica de Turing consistió en abstraer el conjunto de operaciones que efectúa un ser humano cuando efectúa un cálculo, de la intención que lo lleva a realizarlo. De esta manera, limitó las habilidades del calculista, en todo momento, a: (1) cambiar un símbolo escrito en un cuadrado de un papel, (2) cambiar su «estado mental», y (3) desplazar su mirada al cuadrado de la derecha o de la izquierda. Turing argumentó que, en estas condiciones, el calculista podía ser reemplazado por una máquina. Lo interesante es que este modelo tan simple no tiene menos poder computacional que las computadoras (a menos que se refute la tesis de Church).
Primero se presenta un modelo inicial de máquinas de Turing (Sección 1.1.1), y luego algunos modelos equivalentes de la gran variedad existente (Sección 1.1.2). En particular, las máquinas de Turing con varias cintas y las máquinas de Turing no determinísticas, serán fundamentales para el desarrollo de la segunda parte del libro. En el Capítulo A1.1 del Apéndice 1 se presenta un modelo equivalente adicional, las máquinas RAM, muy similares estructuralmente a las computadoras.
Contar con distintos modelos de máquinas de Turing facilitará las demostraciones de los teoremas y la presentación de ejemplos. Las pruebas de equivalencia entre modelos de máquinas de Turing son inductivas, y se basan en la técnica de simulación.

1.1.1. Modelo inicial de máquinas de Turing

Una *máquina de Turing* (o MT) M está compuesta por:

- Una cinta infinita en los dos extremos, dividida en celdas. Cada celda puede almacenar un símbolo.
- Una unidad de control. En todo momento la unidad de control almacena el *estado corriente* de M.
- Un cabezal. En todo momento el cabezal apunta a una celda. El símbolo apuntado se denomina *símbolo corriente*. El cabezal puede moverse sólo de a una celda por vez, a la izquierda o a la derecha.

La figura 1.1.1 muestra los componentes de una máquina de Turing.

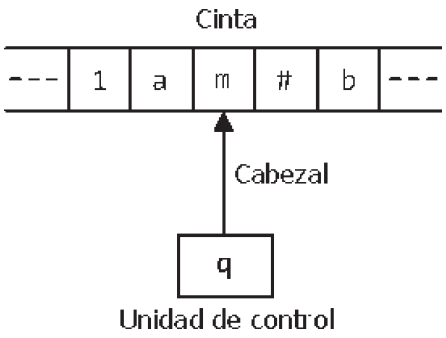


Figura 1.1.1

Los estados pertenecen a un conjunto finito Q, y los símbolos, a un conjunto finito Γ.
Al comienzo, en la *configuración (*o *instancia) inicial* de M, la cinta tiene el *string inicial* o *input*, limitado a izquierda y derecha por infinitos símbolos blancos (que se denotan con «B»). La unidad de control almacena el *estado inicial* q_0. Y el cabezal apunta al primer símbolo del input, es decir a su símbolo de más a la izquierda (si el input es el string vacío λ, entonces el cabezal apunta a algún «B»).
A partir de la configuración inicial, M se comporta de acuerdo a lo especificado en su *función de transición* δ. M en cada paso lee un estado y un símbolo, eventualmente los modifica, y se mueve un lugar a la derecha o a la izquierda o no se mueve. Cuando δ no está definida para el estado corriente y el símbolo corriente, M se detiene (si esto nunca sucede, entonces M nunca se detiene, «tiene un *loop*»).
La definición formal de una máquina de Turing es la siguiente:

Definición. Una MT M es una 6-tupla $M = <Q, \Sigma, \Gamma, \delta, q_0, F>$, tal que:

- Q es el conjunto de estados de M. Es un conjunto finito.
- Σ es el alfabeto de los inputs de M. Es un conjunto finito.
- Γ es el alfabeto de los strings de la cinta de M. Es un conjunto finito, e incluye estrictamente a Σ. Por convención, «B» ε $(\Gamma - \Sigma)$.

- δ es la función de transición de M. Se define δ: Q x $\Gamma \rightarrow$ Q x Γ x {L, R, S}, tal que L representa el movimiento del cabezal a la izquierda, R el movimiento a la derecha, y S indica que el cabezal no se mueve.
- q_0 es el estado inicial de M. Se cumple que $q_0 \in Q$.
- Finalmente, F es el conjunto de estados finales de M. Se cumple que $F \subseteq Q$.

□

Si a partir del input w, la MT M se detiene en un estado $q \in F$, se dice que M *acepta* w. En cambio, cuando a partir de w, la MT M se detiene en un estado $q \in (Q - F)$ o no se detiene, se dice que M *no acepta* (o *rechaza*) w. El conjunto de los strings aceptados por la MT M es el lenguaje aceptado o reconocido por M, y se denota con L(M).

Ejemplo. Sea el lenguaje $L = \{a^n b^n \mid n \geq 1\}$. Es decir, L es un lenguaje infinito formado por los strings «ab», «aabb», «aaabbb», etc. Se va a construir una MT M que acepte L, o en otras palabras, tal que L(M) = L.

Idea general: por cada símbolo «a» que M lee, lo reemplaza por el símbolo «α» y va a la derecha hasta encontrar el primer símbolo «b». Cuando lo detecta, lo reemplaza por el símbolo «β» y vuelve a la izquierda para repetir el proceso a partir de la «a» inmediatamente a la derecha de la anterior. Si al final del proceso no sobran «a» ni «b», entonces M se detiene en un estado de F; en caso contrario se detiene en un estado de (Q – F).

Formalmente, sea la MT M = <Q, Σ, Γ, δ, q_0, F>, tal que:

- $Q = \{q_a, q_b, q_L, q_H, q_F\}$. El estado q_a es el estado en que M busca una «a». El estado q_b es el estado en que M busca una «b». El estado q_L es el estado en que M vuelve a la izquierda para procesar la siguiente «a». El estado q_H es el estado en que M detecta que no hay más «a». Por último, el estado q_F es el estado de aceptación (M detecta que no hay más «b»).
- $\Sigma = \{a, b\}$.
- $\Gamma = \{a, b, \alpha, \beta, B\}$.
- $q_0 = q_a$.
- $F = \{q_F\}$.
- De acuerdo a la idea general descripta anteriormente, la función de transición δ se define de la siguiente manera:

1) $\delta(q_a, a) = (q_b, \alpha, R)$.
2) $\delta(q_b, a) = (q_b, a, R)$.
3) $\delta(q_b, b) = (q_L, \beta, L)$.
4) $\delta(q_L, \beta) = (q_L, \beta, L)$.
5) $\delta(q_L, a) = (q_L, a, L)$.
6) $\delta(q_L, \alpha) = (q_a, \alpha, R)$.
7) $\delta(q_b, \beta) = (q_b, \beta, R)$.
8) $\delta(q_a, \beta) = (q_H, \beta, R)$.
9) $\delta(q_H, \beta) = (q_H, \beta, R)$.
10) $\delta(q_H, B) = (q_F, B, S)$.

La función de transición δ se puede describir alternativamente en forma tabular, indicando los estados q en las filas y los símbolos x en las columnas, de modo tal que en los cruces aparezca la terna asociada a $\delta(q, x)$. Más adelante, en algún ejemplo se utilizará este tipo de descripción.
La construcción de la MT M anterior parece correcta. Puede verificarse, por ejemplo, que M acepta el input w = «aaabbb» después de varios pasos, y que rechaza el input w = «a» casi al comienzo. La prueba formal de que M reconoce el lenguaje L consiste en demostrar por doble inclusión de conjuntos que L(M) = L :

$w \varepsilon L \leftrightarrow$ w no es el string vacío λ y tiene símbolos «a» seguidos de símbolos «b» en igual cantidad $\leftrightarrow$ a partir del input w, la MT M se detiene en su estado $q_F \leftrightarrow w \varepsilon L(M)$.

Por convención, una configuración C de una MT M en la que el estado corriente es q, la cinta contiene los símbolos … $Bx_1x_2x_3 \ldots x_i \ldots x_kB$ …, y el cabezal apunta al símbolo x_i, se representa de la siguiente manera:

$$x_1x_2x_3 \ldots qx_i \ldots x_k \, .$$

Es decir, se escribe el estado corriente a la izquierda del símbolo corriente.
La expresión $C_1 \vdash_M C_2$ denota que la MT M pasa en un paso de la configuración C_1 a la configuración C_2. Y la expresión $C_1 \vdash^*_M C_2$, que M pasa en cero o más pasos de C_1 a C_2.
De esta manera, considerando el ejemplo anterior con el input «aaabbb», se puede escribir:

$$q_aaaabbb \vdash_M \alpha q_baabbb \vdash_M \alpha aq_babbb \vdash_M \ldots,$$

siendo el lenguaje reconocido por la MT M:

$L(M) = \{w \mid (w \in \Sigma^*) \wedge (\exists w_1, w_2 \in \Gamma^*, \exists q \in F : q_0 w \vdash_M^* w_1 q w_2) \wedge (\delta$ no está definida para el estado q y el primer símbolo del string $w_2)\}$.

1.1.2. Modelos equivalentes de máquinas de Turing

Modificando determinadas características de las MT del modelo definido previamente (por ejemplo, aumentando la cantidad de cintas, restringiendo los movimientos del cabezal, acotando la cantidad de estados, etc.), se obtienen otros modelos de MT. Será de utilidad valerse de distintos modelos de MT, como se verá en lo que sigue.
Se define que dos modelos son *equivalentes* cuando sus MT tienen el mismo poder computacional. Más precisamente, si dos MT M_1 y M_2 son equivalentes cuando $L(M_1) = L(M_2)$, se define que el modelo $\mathcal{M}_1$ de MT es equivalente al modelo $\mathcal{M}_2$ de MT cuando para toda MT M_1 de $\mathcal{M}_1$ existe una MT M_2 equivalente de $\mathcal{M}_2$ y para toda MT M_3 de $\mathcal{M}_2$ existe una MT M_4 equivalente de $\mathcal{M}_1$.
Los modelos de MT que se presentan a continuación son equivalentes al modelo inicial, y serán utilizados frecuentemente a lo largo de los próximos capítulos.

1.1.2.1. Modelo de máquinas de Turing con estados q_A y q_R

Las MT de este modelo tienen dos estados especiales, uno de aceptación y otro de rechazo. Cuando las MT se detienen, lo hacen sólo en alguno de ellos.

Definición. Una MT M con estados q_A y q_R se define de la siguiente manera:

$M = <Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R>$, tal que las características de M son las mismas que las de una MT del modelo inicial, salvo que:

- Sus estados q_A y q_R no pertenecen al conjunto Q.
- Su función de transición se define como δ: $Q \times \Gamma \rightarrow (Q \cup \{q_A, q_R\}) \times \Gamma \times \{L, R, S\}$.
- Si M se detiene, lo hace sólo en q_A ó q_R.

□

M acepta un input w cuando a partir de él se detiene en el estado q_A .Si en cambio se detiene en el estado q_R o no se detiene, M rechaza w. Por lo tanto, ahora:

$$L(M) = \{w \mid (w \varepsilon \Sigma^*) \wedge (\exists w_1, w_2 \varepsilon \Gamma^* : q_0 w \vdash^*_M w_1 q_A w_2)\}.$$

A este modelo se lo denominará *modelo estándar*.

Ejemplo. Sea el mismo lenguaje de antes, $L = \{a^n b^n \mid n \geq 1\}$. Se cumple que la siguiente MT del modelo estándar $M = <Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R>$ reconoce L:

- $Q = \{q_a, q_b, q_L, q_H\}$.
- $\Sigma = \{a, b\}$.
- $\Gamma = \{a, b, \alpha, \beta, B\}$.
- $q_0 = q_a$.
- La función de transición δ de M se presenta en forma tabular en la figura 1.1.2, y se basa en la misma idea general de antes.

δ	a	b	α	β	B
q_a	(q_b,α,R)			(q_H,β,R)	
q_b	(q_b,a,R)	(q_L,β,L)		(q_b,β,R)	
q_L	(q_L,a,L)		(q_a,α,R)	(q_L,β,L)	
q_H				(q_H,β,R)	(q_A,B,S)

Figura 1.1.2

En el modelo estándar, la función de transición δ debe estar definida para todo estado $q \varepsilon Q$ y para todo símbolo $x \varepsilon \Gamma$. Por convención, las celdas vacías de la tabla representan ternas con el estado de rechazo.

Los siguientes dos teorema prueban que el modelo estándar es equivalente al modelo inicial.

Teorema. Sea M_1 una MT del modelo inicial. Existe una MT M_2 del modelo estándar equivalente a M_1.

Prueba. Sea M_1 una MT del modelo inicial, con $M_1 = <Q, \Sigma, \Gamma, \delta_1, q_0, F>$. Se va a construir una MT M_2 del modelo estándar equivalente a M_1. Sea $M_2 = <Q, \Sigma, \Gamma, \delta_2, q_0, q_A, q_R>$, tal que $\forall q \in Q$, y $\forall x \in \Gamma$:

1) Si $\delta_1(q, x)$ está indefinida y $q \in F$, entonces $\delta_2(q, x) = (q_A, x, S)$.
2) Si $\delta_1(q, x)$ está indefinida y $q \notin F$, entonces $\delta_2(q, x) = (q_R, x, S)$.
3) Si $\delta_1(q, x)$ está definida, entonces $\delta_2(q, x) = \delta_1(q, x)$.

Se cumple que $L(M_1) = L(M_2)$:

Si $w \in L(M_1)$, entonces $q_0w \vdash^*_{M1} w_1qw_2$, con $q \in F$ y δ_1 indefinida con respecto a q y el primer símbolo de w_2. Por (3), $q_0w \vdash^*_{M2} w_1qw_2$, y por (1), $w_1qw_2 \vdash_{M2} w_1q_Aw_2$. Por lo tanto, $q_0w \vdash^*_{M2} w_1q_Aw_2$, es decir, $w \in L(M_2)$. Se llegó a $L(M_1) \subseteq L(M_2)$.

Si $w \notin L(M_1)$, entonces (a): $q_0w \vdash^*_{M1} w_1qw_2$, con $q \notin F$ y δ_1 indefinida con respecto a q y el primer símbolo de w_2, o bien (b): M_1 no se detiene a partir de la configuración inicial q_0w. En el caso (a), por (3), $q_0w \vdash^*_{M2} w_1qw_2$, y por (2), $w_1qw_2 \vdash_{M2} w_1q_Rw_2$, entonces $q_0w \vdash^*_{M2} w_1q_Rw_2$, es decir $w \notin L(M_2)$. En el caso (b), por (3) M_2 no se detiene a partir de la configuración inicial q_0w, entonces otra vez $w \notin L(M_2)$. Por lo tanto, en (a) y en (b) se llegó a $L(M_2) \subseteq L(M_1)$.

Por doble inclusión se probó $L(M_1) = L(M_2)$.

□

Teorema. Sea M_1 una MT del modelo estándar. Existe una MT M_2 del modelo inicial equivalente a M_1.

Prueba. Sea M_1 una MT del modelo estándar, con $M_1 = <Q_1, \Sigma, \Gamma, \delta, q_0, q_A, q_R>$. Se va a construir una MT M_2 del modelo inicial equivalente a M_1. Sea $M_2 = <Q_2, \Sigma, \Gamma, \delta, q_0, F>$, tal que $Q_2 = Q_1 \cup \{q_A, q_R\}$, y $F = \{q_A\}$.

Se cumple que $L(M_1) = L(M_2)$:

Si $w \in L(M_1)$, entonces $q_0w \vdash^*_{M1} w_1q_Aw_2$, y por construcción $q_0w \vdash^*_{M2} w_1q_Aw_2$. Como $q_A \in F$, y δ está indefinida con respecto a q_A y el primer símbolo de w_2, entonces $w \in L(M_2)$. Se llegó a $L(M_1) \subseteq L(M_2)$.

Si $w \notin L(M_1)$, entonces (a): $q_0w \vdash^*_{M1} w_1q_Rw_2$, o bien (b): M_1 no se detiene a partir de la configuración inicial q_0w. En el caso (a), por construcción $q_0w \vdash^*_{M2} w_1q_Rw_2$, y como $q_R \notin F$ y δ está indefinida con respecto a q_R y el primer símbolo de w_2, entonces $w \notin L(M_2)$. En el caso (b), por construcción, M_2 no se detiene a partir de la configuración inicial q_0w, entonces otra vez $w \notin L(M_2)$. Por lo tanto, en (a) y en (b) se llegó a $L(M_2) \subseteq L(M_1)$.

Por doble inclusión se probó $L(M_1) = L(M_2)$.

□

1.1.2.2. Modelo de máquinas de Turing con K cintas

Las MT de este modelo tienen una o más cintas (una cantidad finita $K \geq 1$). Aunque no resulta intuitivo, su poder computacional es el mismo que el de los modelos anteriores.
Por cada cinta existe un cabezal, y sigue habiendo una sola unidad de control. La figura 1.1.3 muestra los componentes de una máquina de Turing con K cintas.

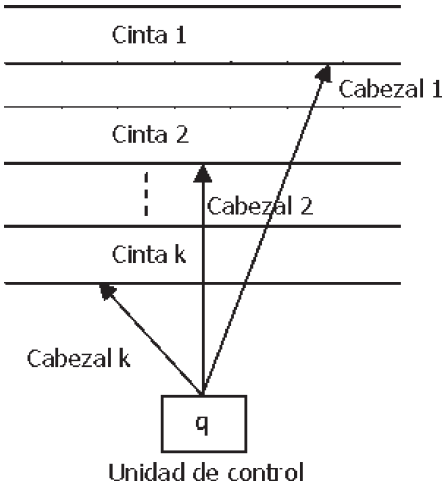


Figura 1.1.3

En la configuración inicial, la primera cinta (cinta 1) contiene el input, la unidad de control almacena el estado q_0, el cabezal de la cinta 1 (cabezal 1) apunta al primer símbolo del input, y el resto de los cabezales apuntan a alguna celda de las cintas correspondientes (las cintas 2 en adelante tienen al comienzo sólo símbolos «B»).
Luego, la MT se comporta de acuerdo a su función de transición δ, que ahora en cada paso lee un estado y una K-tupla de símbolos (los apuntados, respectivamente, por los cabezales 1 a K), y (a) modifica eventualmente el estado, (b) modifica cero, uno o más símbolos, (c) se mueve en cada cinta un lugar a la derecha, a la izquierda o no se mueve. En cada cinta la MT se comporta de manera independiente, es decir que en una cinta puede modificar un símbolo y moverse a la derecha, en otra puede mantener el símbolo corriente y moverse a la izquierda, etc. Formalmente:

Definición. Una MT M con K cintas se define de la siguiente manera:

$M = <Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R>$, tal que las características de M son las mismas que las de una MT del modelo estándar, salvo que $\delta: Q \times \Gamma^K \rightarrow (Q \cup \{q_A, q_R\}) \times (\Gamma \times \{L, R, S\})^K$.

□

Si M se detiene lo hace sólo en los estados q_A ó q_R. Si M se detiene en q_A acepta el input. Y si se detiene en q_R o no se detiene, lo rechaza.

Ejemplo. Sea $L = \{w \mid w \in \{a, b\}^*$ y w es un palíndromo$\}$, donde w es un palíndromo sii $w = w^R$, siendo w^R el string inverso de w. Se va a construir una MT M con K cintas que acepte L.

La idea general de cómo trabaja la MT M, que tendrá 2 cintas, es la siguiente:

1) Copiar el input, que está en la cinta 1, en la cinta 2.
2) Apuntar al primer símbolo del input y al último símbolo del string de la cinta 2.
3) Comparar los símbolos apuntados. Si son distintos, rechazar. Si son iguales y «B», aceptar. En otro caso, moverse a la derecha en la cinta 1, moverse a la izquierda en la cinta 2, y volver al paso (3).

Formalmente, sea $M = <Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R>$, con:

- $Q = \{q_0, q_1, q_2\}$. El estado q_0 es el estado de copia de la cinta 1 a la cinta 2. El estado q_1 es el estado de posicionamiento inicial de los cabezales 1 y 2. Y el estado q_2 es el estado de comparación de los strings de las cintas 1 y 2.
- $\Sigma = \{a, b\}$.
- $\Gamma = \{a, b, B\}$.
- $q_0 = q_0$.
- La función de transición δ se define tabularmente en la figura 1.1.4, en base a lo descripto en la idea general.

δ	a , a	a , b	a , B	b , a	b , b	b , B	B , a	B , b	B , B
q_0			q_0,a,R,a,R			q_0,b,R,b,R			q_1,B,L,B,L
q_1	q_1,a,L,a,S	q_1,a,L,b,S		q_1,b,L,a,S	q_1,b,L,b,S		q_2,B,R,a,S	q_2,B,R,b,S	q_2,B,S,B,S
q_2	q_2,a,R,a,L	q_R,a,S,b,S		q_R,b,S,a,S	q_2,b,R,b,L				q_A,B,S,B,S

Figura 1.1.4

En la tabla, $\delta[q_i, x_1, x_2] = q'_i, x'_1, d_1, x'_2, d_2$, con $d_i \varepsilon \{L, R, S\}$, debe entenderse como $\delta(q_i, (x_1, x_2)) = (q'_i, ((x'_1, d_1), (x'_2, d_2)))$, que para simplificar la escritura también se puede expresar $\delta(q_i, x_1, x_2) = (q'_i, x'_1, d_1, x'_2, d_2)$.

Ejercicio. *Resolver el ejemplo anterior mediante una MT del modelo estándar.*

□

La MT M construida en el ejemplo consume unos 3n pasos (n es el tamaño del input), unos n pasos en cada uno de los incisos descriptos. Se podría construir una MT M' un poco más eficiente, definiendo directamente $\delta(q_1, B, B) = (q_A, B, S, B, S)$, dado que la única posibilidad para esta situación es que el input sea λ. La complejidad computacional medida en los pasos que efectúa una MT, es decir su tiempo de trabajo, será tratada en la segunda parte del libro, en que será muy útil el uso del modelo de máquinas de Turing con K cintas.
Podría suponerse que una MT con varias cintas es capaz de reconocer más lenguajes que una MT con una sola cinta. Sin embargo, se probará a continuación que el modelo de MT con K cintas y el modelo estándar son equivalentes.

Una MT del modelo estándar es un caso particular de MT con K cintas (K = 1). Por lo tanto, la prueba de equivalencia se reduce al siguiente teorema:

Teorema. Sea M_1 una MT del modelo con K cintas. Existe una MT M_2 del modelo estándar equivalente a M_1.

Prueba. Sólo se describe la idea general de la prueba.

Primeramente se introducirá la noción de *track* (o pista). En el caso más general, se puede asumir que toda celda de una MT está dividida en T tracks, con $T \geq 1$ (hasta el momento sólo se ha trabajado con celdas de un track). De este modo, el contenido de una celda puede representarse como una T-tupla $<x_1, x_2, x_3, \ldots, x_T>$, y las 5-tuplas de la función de transición δ tienen la forma
$(q_i, <x_1, x_2, x_3, \ldots, x_T>, q'_i, <x'_1, x'_2, x'_3, \ldots, x'_T>, d)$, con $d \in \{L, R, S\}$.

Sea M_1 una MT del modelo con K cintas. Se va a construir una MT M_2 del modelo estándar con una cinta, equivalente a M_1.

La cinta de M_2 tiene 2K tracks. Los primeros dos tracks representan la cinta 1 de M_1, los siguientes dos tracks representan la cinta 2 de M_1, y así sucesivamente. Dado un par de tracks que representan una cinta de M_1, el primer track almacena en sus distintas sub-celdas el contenido de las celdas correspondientes de M_1, mientras que en el segundo track hay «B» en todas las sub-celdas salvo en una que lleva la marca «X», para indicar que la celda representada de M_1 está apuntada por un cabezal. La tabla de la figura 1.1.5 ejemplifica la representación, para el caso de 3 cintas y por lo tanto 6 tracks:

	celda i	**celda i+1**	**celda i+2**	**celda i+3**	**celda i+4**	**celda i+5**
track 1	- - -	x_1	x_4	x_7	x_{10}	- - -
track 2		**X**				
track 3	- - -	x_2	x_5	x_8	x_{11}	- - -
track 4					**X**	
track 5	- - -	x_3	x_6	x_9	x_{12}	- - -
track 6				**X**		

Figura 1.1.5

De acuerdo a este ejemplo, la celda i+1 de la cinta 1 de M_1 tiene el símbolo x_1 y está apuntada por el cabezal 1, la celda i+2 de la cinta 2 de M_1 tiene el símbolo x_5 y no está apuntada por el cabezal 2, etc.

Si el input de M_1 es $w = w_1 w_2 w_3 \ldots w_n$, entonces al comienzo el input de M_2 es
$\langle w_1, B, B, \ldots, B\rangle \langle w_2, B, B, \ldots, B\rangle \langle w_3, B, B, \ldots, B\rangle \ldots \langle w_n, B, B, \ldots, B\rangle$,
y el cabezal apunta al símbolo $\langle w_1, B, B, \ldots, B\rangle$.

M_2 empieza transformando el primer símbolo de su input en el símbolo $\langle w_1, X, B, X, \ldots, B, X\rangle$, y pasando a un estado que simula el estado inicial de M_1. Luego, cada paso de M_1 es simulado por un conjunto de pasos de M_2, primero de izquierda a derecha y luego de derecha a izquierda:

1) Al inicio de este conjunto de pasos de M_2, su cabezal apunta a la celda con la marca «X» de más a la izquierda.
2) Luego M_2 se mueve a la derecha, memorizando por medio de sus estados los símbolos que están acompañados por una marca y a qué cinta están asociados. Por ejemplo, si se reconoce un símbolo x_i acompañado por una marca correspondiente a la cinta k, el estado corriente de M_2 tendrá un índice con el par (k, x_i). El estado corriente de M_2 memoriza además el número de marcas que quedan por detectar, que se actualiza cuando se encuentra una nueva marca.
3) Cuando M_2 reconoce todas las marcas, emprende la vuelta a la izquierda hasta que se encuentra otra vez con la marca de más a la izquierda, y se va comportando de acuerdo al estado de M_1 que está siendo simulado. Toda vez que encuentra una marca modifica eventualmente (a) el símbolo asociado que representa el contenido de M_1, y (b) la ubicación de la marca, según la información obtenida en el camino de ida y la función de transición de M_1. M_2 otra vez se vale de un contador para detectar cuántas marcas le quedan por recorrer en el camino de vuelta, que actualiza correspondientemente.
4) Finalmente, M_2 modifica su estado acorde a cómo lo modifica M_1. Si en particular el estado de M_1 es de aceptación, entonces M_2 se detiene y acepta, y si es de rechazo, se detiene y rechaza.

□

Ejercicio. *Construir otra vez una MT del modelo estándar que reconozca $L = \{w \mid w \in \{a, b\}^*$ y w es un palíndromo$\}$, pero ahora tomando como*

base la solución anterior con una MT con dos cintas y la simulación utilizada en la prueba del teorema.

□

Notar que las pruebas de equivalencia de modelos de MT que se presentaron, se basan en la simulación de una MT por otra. Dada una MT M_1, se construye una MT M_2 que simula con uno o más pasos cada paso de M_1. En algunos casos, antes de la simulación, la MT M_2 realiza pasos preliminares para alcanzar una configuración que simule la configuración inicial de M_1 (como sucedió en la última prueba). Una vez que se construye la MT M_2, debe probarse que se cumple $L(M_1) = L(M_2)$.

Notar además, con respecto a la última prueba, que como luego de h pasos de la MT M_1 sus cabezales pueden distanciarse a lo sumo 2h celdas, entonces a la MT M_2 le va a llevar simular h+1 pasos de M_1 a lo sumo unos $(4 + 8 + 12 + \ldots + 4h) = 4(1 + 2 + 3 + \ldots + h) = 2h(h+1) = O(h^2)$ pasos. Esto significa que a pesar de que el poder computacional de una MT con K cintas no es mayor que el de una MT con una cinta, reconocer un lenguaje mediante una MT con una cinta llevaría más tiempo (es decir, requeriría más pasos) que mediante una MT equivalente con varias cintas. El tiempo de retardo que se obtuvo es del orden cuadrático en el peor caso. Esta relación se utilizará en la segunda parte del libro.

1.1.2.3. Modelo de máquinas de Turing no determinísticas

En este modelo, a partir del estado corriente q y el símbolo corriente x, una MT puede continuar de varias maneras, no de una sola como se vio hasta ahora. La elección de por cuál terna (q', x', d) la MT continúa es no determinística. Por lo tanto, ahora se hablará de una *relación de transición* Δ en lugar de una función de transición δ.
Como en el caso de las MT con K cintas con respecto a las MT de los modelos inicial y estándar, aunque no resulta intuitivo, el poder computacional de una MT *no determinística* (o MTN) no es mayor que el de las otras, que se caracterizan por ser entonces MT *determinísticas* (o MTD).
Una manera de interpretar cómo trabaja una MTN M es suponer que todas sus computaciones (secuencias de pasos) se ejecutan en paralelo. M aceptará un input w si a partir de w al menos una computación de M se detiene en el estado q_A. En caso contrario, es decir si las com-

putaciones de M son finitas y terminan en el estado q_R o bien son infinitas, M rechazará el input.
La figura 1.1.6 ilustra el posible comportamiento de una máquina de Turing no determinística M.

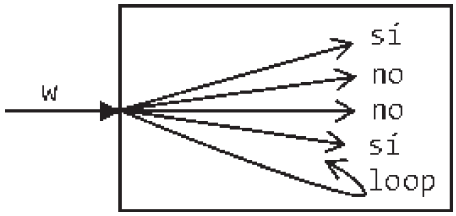


Figura 1.1.6

En la figura, los términos «sí», «no» y «loop», indican que la computación correspondiente se detiene en q_A, en q_R, o no se detiene, respectivamente. En este caso, la MTN M acepta el input w, porque al menos una de sus computaciones se detiene en el estado de aceptación. Formalmente:

Definición. Una MTN M se define de la siguiente manera:

$M = <Q, \Sigma, \Gamma, \Delta, q_0, q_A, q_R>$, tal que las características de M son las mismas que las de una MT del modelo estándar, salvo que ahora M tiene una relación de transición que se define como:

$$\Delta: Q \times \Gamma \rightarrow \mathcal{P}((Q \cup \{q_A, q_R\}) \times \Gamma \times \{L, R, S\}),$$

siendo $\mathcal{P}((Q \cup \{q_A, q_R\}) \times \Gamma \times \{L, R, S\})$ el conjunto de partes de $(Q \cup \{q_A, q_R\}) \times \Gamma \times \{L, R, S\}$, dado que por cada par de $Q \times \Gamma$ puede haber más de una terna definida de $(Q \cup \{q_A, q_R\}) \times \Gamma \times \{L, R, S\}$.

□

La cantidad máxima de ternas asociadas por la relación Δ a un mismo par (q, x) se denomina *grado* de Δ. Por convención, la expresión $\Delta(q, x) = \varnothing$ significa que dado el par (q, x), la MTN se detiene en su estado q_R. M acepta un input w sii lo acepta en al menos una de sus computaciones.

Ejemplo. Sea M = <Q, Σ, Γ, Δ, q_0, q_A, q_R>, con:

- Q = {q_0, q_1, q_2}.
- Σ = {a, b, c}.
- Γ = {a, b, c, B}.
- $q_{0 =} q_{0.}$
- La relación de transición Δ se define de la siguiente manera:

1) $\Delta(q_0, a) = \{(q_1, a, R), (q_2, a, R)\}$.
2) $\Delta(q_0, x) = \{(q_R, x, S)\}$, con x = «b», «c» ó «B».
3) $\Delta(q_1, b) = \{(q_A, b, S)\}$.
4) $\Delta(q_1, x) = \{(q_R, x, S)\}$, con x = «a», «c» ó «B».
5) $\Delta(q_2, c) = \{(q_A, c, S)\}$.
6) $\Delta(q_2, x) = \{(q_R, x, S)\}$, con x = «a», «b» ó «B».

Se prueba fácilmente que L(M) = {w | w ε {a, b, c}$^+$, tal que los strings w comienzan con el substring «ab» o con el substring «ac»}.

Al igual que en el caso de las MTD con K cintas, la utilización de las MTN será muy importante cuando se trate la complejidad computacional en la segunda parte del libro.

Se prueba a continuación que una MTN tiene el mismo poder computacional que una MT del modelo estándar. Como una MT del modelo estándar es un caso particular de MTN tal que el grado de Δ es 1, sólo debe demostrarse lo siguiente:

Teorema. Sea M_1 una MTN. Existe una MT M_2 del modelo estándar equivalente a M_1.

Prueba. Se va a describir únicamente la idea general de la prueba.

Para todo estado y todo símbolo de la MTN M_1, existe un número finito de opciones para el siguiente paso, numeradas como 1, 2, …, K, siendo K el grado de la relación de transición Δ de M_1. De esta manera, se puede representar cualquier computación de M_1 mediante una secuencia de dígitos entre 1 y K, que se denominará *discriminante*. Por ejemplo, la secuencia <3, 4, 1, 3, 2> representa una computación de M_1 en la que en el primer paso se elige la tercera opción de Δ, en el segundo paso la cuarta opción, en el tercero la primera, etc. Algunos discriminantes

pueden representar computaciones no válidas, dado que no siempre tienen que haber K opciones para un par (q, x).
Se va a construir una MTD M_2 con tres cintas, equivalente a la MTN M_1. Por la equivalencia de los modelos de MTD con K cintas y una cinta, entonces por transitividad existirá una MTD M_3 del modelo estándar equivalente a M_1.
La primera cinta de M_2 tiene el input. En la segunda cinta, M_2 genera sistemáticamente secuencias de dígitos entre 1 y K, de menor a mayor longitud y en orden numérico creciente considerando la misma longitud. Primero genera discriminantes de longitud 1 ordenados numéricamente de menor a mayor, después hace lo mismo con los discriminantes de longitud 2, 3, etc. Por ejemplo, los primeros discriminantes son: <1>, <2>, …, <K>, <1, 1>, <1, 2>, …, <1, K>, …, <K, 1>, <K, 2>, …, <K, K>, <1, 1, 1>, … Este orden se denomina *orden canónico*, y será utilizado frecuentemente. Por último, en la tercera cinta M_2 lleva a cabo la simulación de M_1.
Por cada discriminante generado en la cinta 2, M_2 copia el input en la cinta 3 y simula M_1 teniendo en cuenta el discriminante, seleccionando paso a paso las opciones representadas de la relación Δ de M_1. Si existe una opción inválida en el discriminante, M_2 genera el siguiente discriminante según el orden canónico. Si M_1 acepta, entonces M_2 acepta (notar que M_2 solamente acepta o no se detiene).

Se cumple que $L(M_1) = L(M_2)$:

Si $w \varepsilon L(M_1)$, entonces alguna computación de la MTN M_1 acepta w. Por construcción, M_2 aceptará w la vez que el discriminante asociado a dicha computación se genere en su cinta 2. Por lo tanto, $w \varepsilon L(M_2)$, y así, $L(M_1) \subseteq L(M_2)$.

Si $w \varepsilon L(M_2)$, entonces M_2 acepta w, simulando M_1 a partir de w en base a un determinado discriminante que genera alguna vez en su cinta 2. Esto significa que existe una computación de M_1 que acepta w. Por lo tanto, $w \varepsilon L(M_1)$, y así, $L(M_2) \subseteq L(M_1)$.

Por doble inclusión se ha probado que $L(M_1) = L(M_2)$.

Ejercicio. *Describir la función de transición δ de M_2. Considerar, entre otras cosas, (a) la generación del discriminante siguiente, (b) el borrado de la cinta 3 de M_2 antes de una nueva simulación de M_1, y (c) la transformación de las opciones no determinísticas de Δ en opciones determinísticas de δ.*

□

La simulación presentada consiste en recorrer «a lo ancho» el «árbol de computaciones» de M_1 (primero se simula un paso de todas las computaciones, después dos, después tres, etc). Se basa en la técnica de *breadth first search*. Notar que no sirve recorrer el árbol de computaciones de M_1 «a lo largo», es decir rama por rama (instanciación de la técnica de *depth first search*), porque al simular una rama infinita de M_1 antes que una de aceptación, M_2 nunca llegará a aceptar el input correspondiente.
Por otro lado, si K es el grado de la relación de transición Δ de M_1, h pasos de M_1 se van a simular con a lo sumo unos $K + 2K^2 + 3K^3 + \ldots + hK^h = O(K^h)$ pasos de M_2. Esto significa que si bien el poder computacional de una MTN no es mayor que el de una MT estándar, reconocer un lenguaje mediante una MT estándar tardaría más que mediante una MTN equivalente. El tiempo de retardo que se obtuvo es del orden exponencial en el peor caso. Esta relación se utilizará en la segunda parte del libro.
Por último, se puede probar fácilmente (queda como ejercicio para el lector), teniendo en cuenta los teoremas anteriores, que las MTN con K cintas tampoco agregan poder computacional a las MT del modelo estándar.

1.1.2.4. Otros modelos equivalentes de máquinas de Turing

Existen en la literatura numerosos modelos de MT equivalentes a los descriptos anteriormente, como las MT *multi-dimensionales*, las MT *multi-cabezales*, las MT *off-line* (se van a utilizar cuando se trate la complejidad computacional), etc. Entre los ejercicios formulados al final de esta parte del libro, hay varios relacionados con pruebas de equivalencia entre modelos de MT.
En el Apéndice 1 se prueba la equivalencia entre las MT y las máquinas RAM (por *Random Access Machines*), que constituyen otra modelización de las computadoras. Se agrega esta demostración por la similitud en varios sentidos entre la estructura de las RAM y las computadoras (memoria con una palabra inicial, instrucciones como LOAD, STORE, ADD, etc).
En el mismo apéndice, por otro lado, se presentan resumidamente MT restringidas, es decir MT con características limitadas en comparación con el modelo estándar o cualquiera de los modelos equivalentes. La idea es utilizarlas para reconocer tipos de lenguajes específicos, en que no es necesario contar con todo el poder computacional de las MT generales, y así simplificar la tarea de reconocimiento.

Cerrando este capítulo, se presenta un último ejemplo de prueba de equivalencia entre modelos de MT.

Ejemplo. Se cumple que para toda MT $M_1 = <Q_1, \Sigma_1, \Gamma_1, \delta_1, q_0, q_A, q_R>$ con una cinta, existe una MT $M_2 = <\{q\}, \Sigma_2, \Gamma_2, \delta_2, q, q_A, q_R>$ con dos cintas equivalente (la inversa se prueba trivialmente).

Prueba. La idea general de construcción de la MT M_2 con dos cintas a partir de la MT M_1 con una cinta, es simular los estados de M_1 con símbolos de M_2 en su cinta 2.

Formalmente, dado el conjunto $Q_1 = \{q_0, q_1, q_2, \ldots, q_m\}$, $M_2 = <\{q\}, \Sigma_2, \Gamma_2, \delta_2, q, q_A, q_R>$ se define de la siguiente manera:

- $q \neq q_i$ para todo estado q_i de Q_1.
- $\Sigma_2 = \Sigma_1$.
- $\Gamma_2 = \Gamma_1 \cup \{x_0, x_1, x_2, \ldots, x_m\}$, con $x_i \notin \Gamma_1$ para todo símbolo x_i.
- La función de transición δ_2 se define de la siguiente manera:

    1) $\forall x \in \Sigma_2: \delta_2(q, (x, B)) = (q, (x, S), (x_0, S))$.
    2) Si $\delta_1(q_i, x) = (q_k, x', d)$, entonces $\delta_2(q, (x, x_i)) = (q, (x', d), (x_k, S))$, con $d \in \{L, R, S\}$, $q_k \neq q_A$, y $q_k \neq q_R$.
    3) Si $\delta_1(q_i, x) = (q_A, x', d)$ entonces $\delta_2(q, (x, x_i)) = (q_A, (x', d), (x_i, S))$, con $d \in \{L, R, S\}$.
    4) Si $\delta_1(q_i, x) = (q_R, x', d)$ entonces $\delta_2(q, (x, x_i)) = (q_R, (x', d), (x_i, S))$, con $d \in \{L, R, S\}$.

Al comienzo, M_2 hace un paso para escribir en su cinta 2 el símbolo que simula el estado inicial de M_1. Luego M_2 simula paso a paso la MT M_1, haciendo exactamente lo que hace M_1, salvo que en lugar de cambiar de estado para todo estado distinto de q_A y q_R, cambia de símbolo en su cinta 2. Si M_1 se detiene en sus estados q_A ó q_R, entonces M_2 se detiene en sus estados q_A ó q_R, respectivamente. Y si M_1 no se detiene, M_2 no se detiene.

Se cumple que $L(M_1) = L(M_2)$:

$w \in L(M_1) \rightarrow$ a partir de w, la MT M_1 se detiene en su estado q_A (luego de i pasos) $\rightarrow$ a partir de w, la MT M_2 se detiene en su estado q_A (luego de i+1 pasos) $\rightarrow w \in L(M_2)$. Así, $L(M_1) \subseteq L(M_2)$.

$w \notin L(M_1) \rightarrow$ a partir de w, la MT M_1 se detiene en su estado q_R (luego de i pasos) o no se detiene $\rightarrow$ a partir de w, la MT M_2 se detiene en su estado q_R (luego de i+1 pasos) o no se detiene, respectivamente $\rightarrow w \notin L(M_2)$. Así, $L(M_2) \subseteq L(M_1)$.

Por lo tanto, por doble inclusión se probó que $L(M_1) = L(M_2)$.

□

CAPÍTULO 1.2. LENGUAJES RECURSIVAMENTE NUMERABLES Y RECURSIVOS

Introducidas las máquinas de Turing, en este capítulo se trazan las fronteras de la computabilidad y la decidibilidad en términos de los lenguajes reconocidos por las mismas.
En la Sección 1.2.1 se definen los lenguajes recursivamente numerables, para los que existen máquinas de Turing que los reconocen, y los lenguajes recursivos, reconocidos por máquinas de Turing que se detienen a partir de cualquier dato de entrada. Se introduce además la clase de los complementos de los lenguajes recursivamente numerables, y se define su relación con los lenguajes recursivamente numerables y los lenguajes recursivos.
En la Sección 1.2.2 se prueban propiedades de las distintas clases de lenguajes. Para desarrollar las pruebas se utiliza la técnica de combinar máquinas de Turing elementales para obtener máquinas más complejas. Las máquinas de Turing se describen con mayor nivel de abstracción para facilitar la presentación.
El universo de los lenguajes queda dividido básicamente en lenguajes no recursivamente numerables, lenguajes recursivamente numerables no recursivos y lenguajes recursivos. La prueba de la existencia de lenguajes recursivamente numerables que no son recursivos se posterga hasta el capítulo siguiente, con el objeto de introducir recién entonces la relación entre los problemas y los lenguajes, y la visión de máquina de Turing como calculadora de una función.

1.2.1. Conjuntos $\mathcal{L}$, RE, CO-RE y R

Las siguientes definiciones que dan inicio a este capítulo, permiten introducir lo que en este libro se identificará como el *mapa de la computabilidad y decidibilidad*.

Definición. Un lenguaje L es *recursivamente numerable* sii existe una MT M que lo reconoce.

□

Es decir, si $\mathcal{L}$ es el conjunto de todos los lenguajes, sólo los lenguajes recursivamente numerables tienen MT que los reconocen (por eso también se los conoce como lenguajes *computables*). Se probará

después que, en efecto, no todos los lenguajes de $\mathcal{L}$ tienen esta propiedad.
El conjunto o clase de los lenguajes recursivamente numerables se va a denominar RE (por *recursively enumerable languages*). La denominación de recursivamente numerables se debe a que los strings de estos lenguajes se pueden listar sistemáticamente.
Por la definición anterior, entonces, dado L ε RE, si M es una MT tal que L(M) = L, se cumple que:

- $\forall w \varepsilon \Sigma^*$: w ε L $\rightarrow$ M a partir de w se detiene en su estado q_A.
- $\forall w \varepsilon \Sigma^*$: w $\notin$ L $\rightarrow$ M a partir de w se detiene en su estado q_R o no se detiene.

Σ^* es el conjunto de todos los strings finitos compuestos por símbolos de un alfabeto universal Σ. El conjunto universal de lenguajes $\mathcal{L}$ se define como $\mathcal{P}(\Sigma^*)$, es decir que todo lenguaje de $\mathcal{L}$ tiene strings de Σ^*. Las MT tendrán toda vez como input algún string de Σ^*.

Se probará después que sólo algunos lenguajes de RE tienen la propiedad de ser reconocidos por MT que se detienen a partir de cualquier input. La definición siguiente introduce este conjunto de lenguajes.

Definición. Un lenguaje L es *recursivo* sii existe una MT M que lo reconoce, y que se detiene cualquiera sea su input.

□

El conjunto o clase de los lenguajes recursivos se va a denominar R. Estos lenguajes también se conocen como *decidibles*, porque las MT que los reconocen pueden decidir en todos los casos si un input les pertenece o no.
Por la definición anterior, entonces, dado L ε R, si M es una MT tal que L(M) = L, se cumple que:

- $\forall w \varepsilon \Sigma^*$: w ε L $\rightarrow$ M a partir de w se detiene en su estado q_A.
- $\forall w \varepsilon \Sigma^*$: w $\notin$ L $\rightarrow$ M a partir de w se detiene en su estado q_R.

Se cumple por definición que R $\subseteq$ RE. En el desarrollo de este capítulo y el siguiente se probará:
- R $\subset$ RE, es decir, existen lenguajes para los que no hay MT que los reconozcan y se detengan a partir de todos los inputs (en otras palabras, existen lenguajes computables que no son decidibles).

– RE $\subset \mathcal{L}$, es decir, existen lenguajes para los que no hay MT que los reconozcan (en otras palabras, existen lenguajes que no son computables).

La figura 1.2.1 presenta una primera versión del mapa de la computabilidad y decidibilidad.

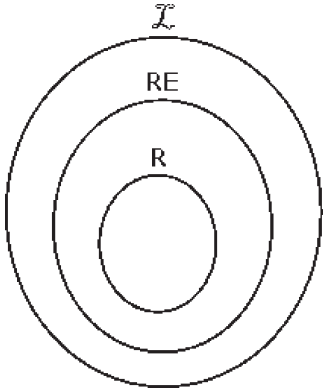


Figura 1.2.1

RE define la barrera entre lo computable y lo no computable, y R define, dentro de lo computable, la barrera entre lo decidible y lo no decidible.

Será también útil definir, y esto quedará claro más adelante, otro conjunto de lenguajes, relacionado con RE y R. Se trata del conjunto CO-RE, formado por los lenguajes complemento (con respecto a Σ^*) de los lenguajes recursivamentes numerables. Formalmente:

Definición. CO-RE = $\{L \mid L \in \mathcal{L} \wedge L^C \in RE\}$, siendo $L^C = \{w \mid w \in (\Sigma^* - L)\}$. □

Se caracterizará después a los lenguajes recursivos como aquéllos que están tanto en RE como en CO-RE. De acuerdo a esta caracterización, la figura 1.2.2 presenta una segunda versión del mapa de la computabilidad y decidibilidad.

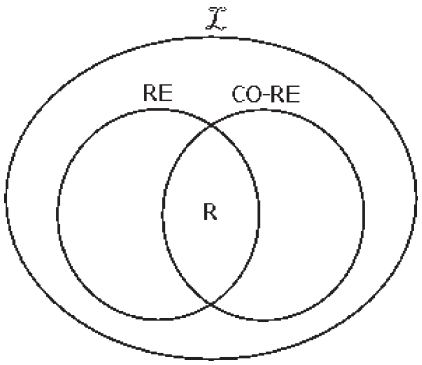


Figura 1.2.2

En lo que sigue, se irá probando la correctitud del mapa. También se presentarán propiedades de las clases RE y R, y se mostrará cómo construir MT combinando MT más simples.

1.2.2. Propiedades de los lenguajes recursivamente numerables y recursivos

El primer teorema de esta sección establece que la clase de lenguajes R es cerrada con respecto a la operación de complemento.

Teorema. Si $L \varepsilon R$, entonces $L^C \varepsilon R$.

Prueba. Sea M una MT que acepta L y se detiene siempre, es decir a partir de cualquier input. Se va a construir una MT M^C que acepta L^C y se detiene siempre. Idea general: dado un input w, si M se detiene en q_A, entonces M^C se detiene en q_R, y viceversa. La figura 1.2.3 ilustra esta idea.

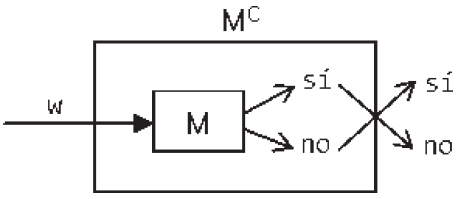


Figura 1.2.3

Formalmente, si M = <Q, Σ, Γ, δ, q_0, q_A, q_R>, entonces M^C = <Q, Σ, Γ, δ', q_0, q_A, q_R>, tal que δ y δ' son idénticas salvo en los casos de aceptación y rechazo:

si $\delta(q, x_i) = (q_A, x_k, d)$, con $q \in Q$, x_i y $x_k \in \Gamma$, $d \in \{L, R, S\}$, entonces $\delta'(q, x_i) = (q_R, x_k, d)$, y
si $\delta(q, x_i) = (q_R, x_k, d)$, entonces $\delta'(q, x_i) = (q_A, x_k, d)$.

M^C se detiene siempre:
Si $w \in L^C \rightarrow w \notin L \rightarrow$ con input w, M se detiene en $q_R \rightarrow$ con input w, M^C se detiene en q_A.
Si $w \notin L^C \rightarrow w \in L \rightarrow$ con input w, M se detiene en $q_A \rightarrow$ con input w, M^C se detiene en q_R.

$L(M^C) = L^C$:
$w \in L(M^C) \leftrightarrow$ con input w, M^C se detiene en $q_A \leftrightarrow$ con input w, M se detiene en $q_R \leftrightarrow w \notin L \leftrightarrow w \in L^C$.

□

Esta es una típica prueba por construcción de que un lenguaje L es recursivo. Se construye una MT M, y se prueba que M se detiene siempre y acepta el lenguaje L. La prueba por construcción de que un lenguaje L es recursivamente numerable, en cambio, requerirá solamente la construcción de una MT M y la prueba de que M acepta L.
El siguiente teorema forma parte de la prueba para caracterizar a los lenguajes recursivos como la intersección de los conjuntos RE y CO-RE.

Teorema. $R \subseteq$ CO-RE.

Prueba. $L \varepsilon R \rightarrow$ por el teorema anterior, $L^C \varepsilon R \rightarrow$ por definición, $L^C \varepsilon RE \rightarrow$ por definición, $L \varepsilon$ CO-RE.

Corolario. Como $R \subseteq RE$ por definición, y $R \subseteq$ CO-RE por el teorema anterior, entonces se cumple $R \subseteq (RE \cap$ CO-RE). Después se probará la inclusión inversa, y así se obtendrá la igualdad mencionada antes: $R = (RE \cap$ CO-RE).

□

El conjunto R también es cerrado con respecto a las operaciones de intersección y unión:

Teorema. Si $L_1 \varepsilon R$ y $L_2 \varepsilon R$, entonces: (a) $(L_1 \cap L_2) \varepsilon R$, y (b) $(L_1 \cup L_2) \varepsilon R$.

Prueba. Se va a probar la parte (a). La parte (b) queda como ejercicio para el lector.

Sean M_1 la MT que acepta L_1 y se detiene siempre, y M_2 la MT que acepta L_2 y se detiene siempre. Se va a construir una MT M que acepta $(L_1 \cap L_2)$ y se detiene siempre.

Idea general: M simulará primero M_1 y luego M_2. Dado un input w, si M_1 se detiene en su estado q_R, entonces directamente M se detiene en su estado q_R. En cambio, si M_1 se detiene en su estado q_A, entonces con el mismo w, la MT M simula M_2, y se detiene en su estado q_A (respectivamente q_R) si M_2 se detiene en su estado q_A (respectivamente q_R). La figura 1.2.4 ilustra esta idea.

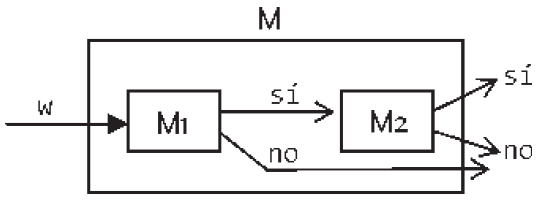


Figura 1.2.4

Formalmente, sea la siguiente MT M con dos cintas. Dado el input w en la cinta 1, M hace:

1) Copiar el input w en la cinta 2.
2) Simular M_1 a partir de w en la cinta 2. Si M_1 se detiene en su estado q_R, entonces M se detiene en su estado q_R.
3) Borrar el contenido de la cinta 2.
4) Copiar el input w en la cinta 2.
5) Simular M_2 a partir de w en la cinta 2. Si M_2 se detiene en su estado q_A (respectivamente q_R), entonces M se detiene en su estado q_A (respectivamente q_R).

Ejercicio. *Completar la prueba, verificando que la MT M construida se detiene siempre, y que* $L(M) = (L_1 \cap L_2)$.

□

Notar que a diferencia de las simulaciones presentadas en el capítulo anterior, las simulaciones de este capítulo consisten directamente en ejecutar una MT M' por parte de otra MT M. Es decir, M «invoca a la subrutina» M', o en otras palabras, la función de transición δ de la MT M incluye un fragmento δ' que no es sino la función de transición de M'. En el capítulo siguiente, en que se trata la máquina de Turing universal, esta noción se generaliza.
El siguiente teorema establece que también la clase RE es cerrada con respecto a las operaciones de unión e intersección.

Teorema. Si $L_1 \in RE$ y $L_2 \in RE$, entonces: (a) $(L_1 \cup L_2) \in RE$, y (b) $(L_1 \cap L_2) \in RE$.

Prueba. Se va a probar la parte (a). La parte (b) queda como ejercicio para el lector.

Sea M_1 la MT que acepta L_1, y M_2 la MT que acepta L_2 (ahora sólo se puede asegurar que estas MT se detienen en los casos de aceptación). Se va a construir una MT M que acepta $(L_1 \cup L_2)$.

Idea general: no sirve que la MT M simule primero M_1 y luego M_2, porque M no aceptará los inputs $w \in (M_2 - M_1)$ a partir de los que M_1 no se detiene. El mismo problema ocurre simulando primero M_2 y después M_1. La solución debe ser otra: consistirá en construir una MT M que

simule «en paralelo» las MT M_1 y M_2, y que acepte si alguna de las dos MT acepta. La figura 1.2.5 ilustra esta idea.

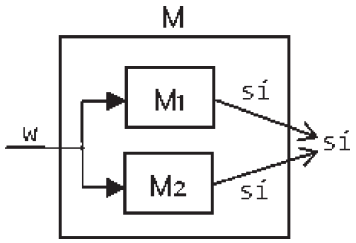


Figura 1.2.5

Formalmente, sea la siguiente MT M con cuatro cintas. Dado el input w en la cinta 1, M hace:

1) Copiar el input w en las cintas 2 y 3.
2) Escribir el número 1 en la cinta 4. Dicho número se va a referenciar como i en lo que sigue.
3) Simular, a partir de w, a lo sumo i pasos de la MT M_1 en la cinta 2, y a lo sumo i pasos de la MT M_2 en la cinta 3. Si M_1 ó M_2 se detienen en q_A, entonces M se detiene en q_A.
4) Borrar el contenido de las cintas 2 y 3.
5) Copiar el input w en las cintas 2 y 3.
6) Sumar 1 al número i de la cinta 4.
7) Volver al paso (3).

Notar que la MT M se detiene en su estado q_A o no se detiene. Puede modificarse la construcción, haciendo que M se detenga en q_R si en el paso (3) se detecta que tanto M_1 como M_2 se detienen en sus estados q_R. Otra mejora en cuanto al tiempo de trabajo de M es no simular cada vez M_1 y M_2 desde el principio.
Las sumas del paso (6) pueden implementarse, por ejemplo, representando el contador i de la cinta 4 en base unaria (1 = «1», 2 = «11», 3 = «111», etc.), de modo tal que sumar 1 significa agregar un «1» a la derecha del string corriente.
En el paso (3) se indica que se simulan a lo sumo i pasos de las MT M_1 y M_2, porque las mismas podrían detenerse antes.

La correctitud de la construcción de la MT M se prueba fácilmente, y queda como ejercicio para el lector.

Ejercicio. *Describir el fragmento de la función de transición δ de M que simula a lo sumo i pasos de M_1 y de M_2.*

□

Finalmente, el siguiente teorema cierra la prueba de la igualdad R = (RE ∩ CO-RE), que es una importante caracterización de la clase R: un lenguaje L es recursivo sii tanto L como su complemento L^C son recursivamente numerables.

Teorema. Si L ε RE y L^C ε RE, entonces L ε R. En otras palabras, (RE ∩ CO-RE) ⊆ R.

Prueba. Sean M la MT que acepta L, y M^C la MT que acepta L^C. Se va a construir una MT M_1 que se detiene siempre y acepta L.

Idea general: M_1 va a simular en paralelo (como se describió previamente) las MT M y M^C. Si M se detiene en q_A, entonces la MT M_1 se detiene en su estado q_A. En cambio, si M^C se detiene en q_A, entonces M_1 se detiene en su estado q_R. La figura 1.2.6 ilustra esta idea.

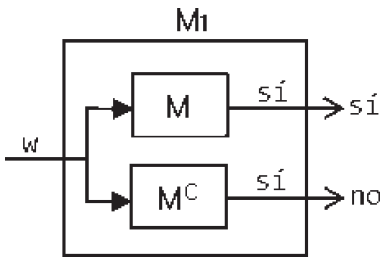


Figura 1.2.6

La construcción de la MT M_1 es muy similar a la de la prueba anterior (no se va a desarrollar, queda como ejercicio para el lector). Se cumple que M_1 se detiene siempre, porque cualquiera sea el input w, w ε L ó w ε L^C, y por lo tanto M ó M^C aceptarán w. También se cumple que $L(M_1)$ = L, porque: w ε $L(M_1)$ ↔ M_1 acepta w ↔ M acepta w ↔ w ε L(M) ↔ w ε L.

Corolario. En un teorema anterior se demostró que $R \subseteq (RE \cap CO\text{-}RE)$, y ahora se ha probado que $(RE \cap CO\text{-}RE) \subseteq R$. Por lo tanto, se cumple $R = (RE \cap CO\text{-}RE)$.

□

Con las definiciones y teoremas anteriores, y asumiendo que $R \subset RE$ (esto se va a probar recién en el capítulo siguiente), se verifica fácilmente la correctitud del mapa de la computabilidad y decidibilidad presentado previamente en la figura 1.2.2:

1) $R \neq \varnothing$. En el capítulo anterior se mostraron ejemplos de lenguajes recursivos. Otro ejemplo de lenguaje recursivo es Σ^*, es decir el lenguaje de todos los strings: una MT M que acepta en el primer paso cualquier símbolo, cumple que se detiene siempre y que $L(M) = \Sigma^*$. También es fácil demostrar que cualquier lenguaje finito es recursivo.

2) $R = (RE \cap CO\text{-}RE)$. Se probó recién.

3) $R \subset RE$. Se va a probar en el capítulo siguiente, para introducir conceptos importantes tales como máquina de Turing universal, problema, función computable, etc. Para probar que se cumple $R \subset RE$, se va a encontrar un lenguaje de $(RE - R)$.

4) $R \subset CO\text{-}RE$. Se probó que $R \subseteq CO\text{-}RE$. Supóngase $R = CO\text{-}RE$. Si $L \in (RE - R)$, entonces vale $L^C \in CO\text{-}RE$ por definición. Como $R = CO\text{-}RE$, entonces $L^C \in R$, y por ser R cerrado con respecto al complemento, se cumple que $L \in R$, lo que contradice la hipótesis $L \in (RE - R)$.

5) $RE \neq CO\text{-}RE$. Si $RE = CO\text{-}RE$, entonces $RE = (RE \cap CO\text{-}RE)$. Por (2), $R = RE$, lo que contradice (3).

6) $RE \subset \mathcal{L}$. Si $L \in (RE - R)$, entonces $L^C \notin RE$, porque de lo contrario, por (2), debe cumplirse que $L \in R$.

7) $CO\text{-}RE \subset \mathcal{L}$. Si $CO\text{-}RE = \mathcal{L}$, entonces $RE = (RE \cap CO\text{-}RE)$. Por (2), $R = RE$, lo que contradice (3).

8) $(RE \cup CO\text{-}RE) \subset \mathcal{L}$. Para probar esto se va a encontrar un lenguaje perteneciente al conjunto $\mathcal{L} - (RE \cup CO\text{-}RE)$:

Sea $L \varepsilon (RE - R)$, y por lo tanto $L^C \notin RE$. Si $L_{01} = \{0y \mid y \varepsilon L\} \cup \{1y \mid y \varepsilon L^C\}$, entonces se cumple que $L_{01} \notin (RE \cup CO\text{-}RE)$:

Vale $L_{01} \notin RE$. Supóngase que $L_{01} \varepsilon RE$. Entonces existe una MT M_{01} que reconoce L_{01}. Se va a construir una MT M que reconoce L^C (absurdo porque $L^C \notin RE$). Dado el input y, M produce el string 1y, luego simula M_{01} a partir de 1y, y responde como M_{01}.
$L^C = L(M)$: $y \varepsilon L^C \leftrightarrow 1y \varepsilon L_{01} \leftrightarrow M_{01}$ acepta $1y \leftrightarrow$ M acepta y.

Vale $L_{01} \notin CO\text{-}RE$. Supóngase que $L_{01} \varepsilon CO\text{-}RE$, o lo que es lo mismo, $L_{01}{}^C \varepsilon RE$. Entonces existe una MT M_{01} que reconoce $L_{01}{}^C$. Se va a construir una MT M que reconoce L^C (absurdo porque $L^C \notin RE$). Dado el input y, M produce el string 0y, luego simula M_{01} a partir de 0y, y responde como M_{01}.
$L^C = L(M)$: $y \varepsilon L^C \leftrightarrow 0y \varepsilon L_{01}{}^C \leftrightarrow M_{01}$ acepta $0y \leftrightarrow$ M acepta y.

□

De este modo se pueden distinguir cuatro categorías de lenguajes, que con dificultad creciente en términos de la computabilidad se enumeran a continuación:

- Lenguajes recursivos o decidibles, es decir el conjunto R.
- Lenguajes recursivamente numerables que no son recursivos, es decir el conjunto (RE – R).
- Lenguajes no recursivamente numerables tales que sus complementos son recursivamente numerables, es decir el conjunto (CO-RE – R).
- Lenguajes no recursivamente numerables tales que sus complementos no son recursivamente numerables, es decir el conjunto $\mathcal{L}$ – (RE ∪ CO-RE).

En este contexto, dado un par cualquiera de lenguajes L y L^C se cumple que:
- L y L^C pertenecen a R, o bien
- $L \varepsilon (RE - R)$ y $L^C \varepsilon$ (CO-RE – R), o bien
- L y L^C pertenecen a ($\mathcal{L}$ – (RE ∪ CO-RE)).

Cuando se estudie la complejidad computacional en la segunda parte del libro, también se establecerán jerarquías de dificultad de lenguajes, pero en términos del espacio y tiempo requeridos por las MT para reconocerlos. Y sólo tendrá sentido hacerlo dentro de la clase R de los lenguajes decidibles.

Para terminar este capítulo, se presentan dos últimos ejemplos de pruebas de propiedades de los lenguajes recursivamente numerables y recursivos.

Ejemplo. La definición inductiva del lenguaje L^k es: (a) $L^0 = \{\lambda\}$, y (b) $L^i = L^{i-1} \bullet L$, con $1 \leq i \leq k$, siendo «•» la operación de concatenación o producto de lenguajes. Por ejemplo, si $L = \{\lambda, ab, c\}$, entonces $L^0 = \{\lambda\}$, $L^1 = L$, $L^2 = \{\lambda, ab, c, abab, abc, cab, cc\}$, etc.

Se cumple que: $L \in R \rightarrow \forall k \geq 0 : L^k \in R$.

Prueba. Se probará por inducción. Sea $L \in R$.

$L^0 = \{\lambda\} \in R$: la MT M_0 que rechaza inmediatamente todo input $w \neq \lambda$ y acepta inmediatamente el input $w = \lambda$, acepta el lenguaje L^0 y se detiene siempre.

Supóngase que $L^m \in R$, con $m \geq 0$. Se va a probar que $L^{m+1} = (L^m \bullet L) \in R$. Sea la siguiente MT M_{m+1}, que a partir del input w en su cinta 1, tal que $|w| = n$, hace lo siguiente (en lo que sigue, M_m es la MT que reconoce el lenguaje L^m, y M_1 es la MT que reconoce el lenguaje L):

1) Escribir el número 0 en la cinta 2. Sea i dicho número.
2) Escribir el número n en la cinta 3. Sea h dicho número.
3) Escribir los primeros i símbolos de w en la cinta 4.
4) Escribir los últimos h símbolos de w en la cinta 5.
5) Simular M_m en la cinta 4 a partir del contenido de dicha cinta, y simular M_1 en la cinta 5 a partir del contenido de dicha cinta. Si ambas simulaciones se detienen en q_A, entonces detenerse en q_A.
6) Si i = n, detenerse en q_R.
7) Hacer i := i + 1 en la cinta 2, y h := h – 1 en la cinta 3, borrar los contenidos de las cintas 4 y 5, y volver al paso (3).

Se cumple que M_{m+1} se detiene siempre, porque M_m y M_1 se detienen siempre.

Se cumple además que $L(M_{m+1}) = L^{m+1}$:
$w \in L(M_{m+1}) \rightarrow$ w es la concatenación de dos strings $w_1 \in L(M_m)$ y $w_2 \in L(M_1) \rightarrow w \in L^{m+1}$.

$w \notin L(M_{m+1}) \rightarrow$ w no es la concatenación de dos strings $w_1 \in L(M_m)$ y $w_2 \in L(M_1) \rightarrow w \notin L^{m+1}$.

Por lo tanto, por inducción se ha probado que $\forall k \geq 0 : L^k \in R$.

□

Ya se probó que la clase de lenguajes RE es cerrada con respecto a la unión, empleando MT determinísticas. Ahora se probará lo mismo de una manera mucho más sencilla, construyendo una MT no determinística.

Ejemplo. Se va a probar de otra manera que si $L_1 \in RE$ y $L_2 \in RE$, entonces $(L_1 \cup L_2) \in RE$.

Prueba. Sean $M_1 = <Q_1, \Sigma_1, \Gamma_1, \delta_1, q_{10}, q_A, q_R>$ y $M_2 = <Q_2, \Sigma_2, \Gamma_2, \delta_2, q_{20}, q_A, q_R>$, dos MTD que aceptan respectivamente los lenguajes L_1 y L_2. Se va a construir una MTN M que acepta $(L_1 \cup L_2)$.

La MTN M se define de la siguiente manera, siendo q_0 un estado que no está en Q_1 ni en Q_2:

$M = <Q = Q_1 \cup Q_2 \cup \{q_0\}, \Sigma = \Sigma_1 \cup \Sigma_2, \Gamma = \Gamma_1 \cup \Gamma_2, \Delta, q_0, q_A, q_R>$, tal que:

$\Delta = \delta_1 \cup \delta_2 \cup \{(q_0, x, q_{10}, x, S), (q_0, x, q_{20}, x, S)\}$, considerando todos los x de Σ.

Es decir, al comienzo la MTN M pasa no determinísticamente a la configuración inicial de la MT M_1 o a la configuración inicial de la MTD M_2, y después se comporta determinísticamente como ellas.

Se cumple que $L_1 \cup L_2 = L(M)$:
$w \in (L_1 \cup L_2) \leftrightarrow w \in L_1$ ó $w \in L_2 \leftrightarrow M_1$ acepta w ó M_2 acepta $w \leftrightarrow$ M acepta $w \leftrightarrow w \in L(M)$.

□

CAPÍTULO 1.3. EL PROBLEMA DE LA DETENCIÓN

En este capítulo se prueba la existencia de un primer lenguaje recursivamente numerable que no es recursivo, lo que completa el desarrollo de las demostraciones asociadas al trazado del mapa de la computabilidad y decidibilidad iniciado en el capítulo anterior.
Primero se introducen conceptos fundamentales como máquina de Turing universal, codificación de una máquina de Turing (Sección 1.3.1), problema de decisión, relación entre problemas y lenguajes, y visión de máquina de Turing como calculadora de una función (Sección 1.3.2).
En la Sección 1.3.3 se desarrolla la prueba referida, correspondiente a un teorema clásico de la teoría de la computación, que establece que el problema de la detención de las máquinas de Turing es semidecidible, o lo que es lo mismo, que el lenguaje asociado al problema de la detención es recursivamente numerable pero no recursivo.
Para la prueba del teorema se introduce la técnica de diagonalización. También por diagonalización se vuelve a probar que hay más lenguajes que lenguajes recursivamente numerables.

1.3.1. Máquina de Turing universal

Para los teoremas y ejemplos que se van a desarrollar en este capítulo y el siguiente, será muy útil trabajar con la visión de *máquina de Turing universal*.
Como modelo de una computadora, es natural considerar que una MT es «programable», es decir que dada una MT U:

- Sus inputs son pares de la forma (<M>, w), siendo el string <M> la codificación de una MT M, y el string w el input de M.
- La ejecución de U consiste en simular la MT M a partir del string w.

Como toda MT, U será una tupla $<Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R>$, con sus propios estados y símbolos, y en particular su función de transición δ establecerá cómo simular la MT M, codificada mediante el string <M>, a partir de w. Esta es la visión de las MT como máquinas universales, dado que pueden ejecutar cualquier MT. Una posible manera de codificar una MT es la siguiente:

Dada $M = <Q, \Sigma, \Gamma, \delta, q_1, q_A, q_R>$:

- Si $Q = \{q_1, q_2, q_3, \ldots, q_k\}$, con $k = |Q|$, los estados de Q se representan por los números 1, 2, 3, …, k, en notación binaria. Además, q_A y q_R se representan en binario por los números k +1 y k + 2, respectivamente.
- Si $\Gamma = \{x_1, x_2, x_3, \ldots, x_k\}$, con $k = |\Gamma|$, los símbolos de Γ se representan por los números 1, 2, 3, …, k, en notación binaria, siendo por convención los símbolos de Σ los primeros $|\Sigma|$ símbolos, y «B» el último símbolo.
- Con respecto a los movimientos d ε {L, R, S}, L se representa por el número 1, R por el 2, y S por el 3, todos en notación binaria.
- Finalmente, la función de transición δ se representa por una secuencia de 5-tuplas, separadas entre sí por el símbolo «#», y los componentes de las 5-tuplas se separan entre sí por el símbolo «**,**».

Con estas consideraciones, la codificación <M> de la MT M consiste en el string formado por |Q|, en notación binaria, seguido por el separador «#» y luego por la representación de la función de transición δ. Se antepone |Q| para facilitar la interpretación de las representaciones de los estados q_A y q_R.

Ejemplo. Si $M = <Q = \{q_1\}, \Sigma = \{x_1\}, \Gamma = \{x_1, B\}, \delta, q_1, q_A, q_R>$, con:

$\delta(q_1, x_1) = (q_A, x_1, S)$, y
$\delta(q_1, B) = (q_R, B, R)$,

entonces el código <M> de la MT M consiste en el siguiente string:

«1#1**,**1**,**10**,**1**,**11#1**,**10**,**11**,**10**,**10».

Dada la MT U con input (<M>, w), su función de transición δ debe interpretar las 5-tuplas codificadas en <M> para simular la MT M a partir del input w. Los símbolos de w se representarán también en notación binaria, separados entre sí por el símbolo «**,**». Para separar <M> de w se utilizará el string «##».

1.3.2. Máquina de Turing para resolver problemas

Siguiendo con la introducción de visiones de MT y conceptos que facilitarán el desarrollo de las próximas secciones, cabe detenerse ahora en la visión de MT para resolver *problemas*.
Hasta el capítulo anterior se ha trabajado con MT que reconocen lenguajes. Otra visión, que es la más general, es la de MT que calculan (o computan) funciones para resolver problemas (entre los que se encuentra en particular el problema de reconocimiento de lenguajes). Por ejemplo, el objetivo de una MT podría ser calcular el cuadrado de un número natural, calcular el perímetro de un triángulo, obtener de un grafo un subgrafo completo o clique de tamaño K, obtener la asignación de valores de verdad con la mínima cantidad de valores verdaderos que satisfaga una fórmula booleana, etc.
En estos casos se deberá distinguir del conjunto de cintas, una *cinta de output* en la que la MT escriba la solución del problema. Correspondientemente, no sólo se deberá tener en cuenta en qué estado termina la MT, sino también cuál es el contenido de su cinta de output.
Existe a su vez una tercera visión de MT, la de las MT que generan lenguajes. Esta visión se presenta suscintamente en el Apéndice 1, en que se muestra cómo se relaciona la visión de MT que generan lenguajes con la visión de MT que reconocen lenguajes.

En un problema se distinguen tres componentes:

- un conjunto $\mathbb{I}$ de *instancias* (por ejemplo, el conjunto de todos los triángulos),
- un conjunto $\mathbb{S}$ de posibles *soluciones* (por ejemplo, los números naturales), y
- una *relación* $\mathbb{R}$, que asocia las instancias del problema con las soluciones (por ejemplo, para el problema de calcular el perímetro de un triángulo, un par <t, n> ε $\mathbb{R}$ sii el número natural n es el perímetro del triángulo t).

Para lo que se pretende presentar en este libro, alcanzará con utilizar un subconjunto propio del universo de problemas, los *problemas de decisión*. Resolver un problema de decisión consiste en, dada una instancia de problema, responder adecuadamente «sí» o «no». Un ejemplo es el problema de decidir si los lenguajes reconocidos por dos MT son iguales. El problema podría enunciarse así: «Dadas las MT M_1 y M_2, ¿acaso $L(M_1) = L(M_2)$?».

La ventaja de considerar este tipo de problemas, además de simplificar la presentación, es que se puede seguir trabajando dentro del marco de la teoría de los lenguajes formales, como se ha venido haciendo hasta ahora. Siguiendo con el ejemplo anterior, el problema de igualdad de lenguajes podría representarse por un lenguaje de la siguiente manera:

$$L = \{(<M_1>, <M_2>) \mid L(M_1) = L(M_2)\},$$

tal que $<M_i>$ es la codificación de la MT M_i. Así, si una MT M reconoce L y se detiene siempre, en realidad estará resolviendo el problema de decisión de igualdad de lenguajes.
En el Apéndice 2 se considera resumidamente otro tipo de problemas, en el marco de la complejidad computacional. En cuanto a lo que sigue, cuando se haga referencia a problemas deberán interpretarse siempre como problemas de decisión, salvo que se indique lo contrario explícitamente.
Luego de esta introducción, ya se está en condiciones de desarrollar, en la siguiente sección, el Teorema del Problema de la Detención (o Teorema del *Halting Problem*), en el que se va a encontrar un lenguaje recursivamente numerable que no es recursivo.

1.3.3. Teorema del Problema de la Detención

El problema de la detención es un problema clásico de la literatura y se enuncia de la siguiente manera: «Dada la MT M y el input w, ¿acaso M se detiene a partir de w?». El lenguaje que representa el problema se denomina HP (por *halting problem*) y es el siguiente:

$$HP = \{(<M>, w) \mid \text{la MT M se detiene a partir del input w}\}.$$

Se prueba, por el Teorema del Problema de la Detención, que $HP \in (RE - R)$, y así que $R \subset RE$.

Primero se van a probar dos lemas, que serán utilizados no sólo en la demostración del teorema sino también en pruebas posteriores.

Lema 1. El lenguaje de los códigos de las MT es recursivo.

Prueba del Lema 1. Se puede construir fácilmente una MT M que reconozca dicho lenguaje y se detenga siempre. Dado un string w, M debe comprobar que:

$w = |Q| \# <\delta>$,

tal que |Q| es un número en binario, <δ> tiene 5-tuplas separadas por «#» de números en binario separados por «**,**», ningún número que representa un estado en <δ> es mayor que |Q| + 2, no hay 5-tuplas repetidas, etc.

□

Lema 2. Existe una MT M que computa, para cualquier número natural i, el código de la MT i-ésima, según el orden canónico.

Prueba del Lema 2. Se puede construir fácilmente una MT M que imprima en su cinta de output el código de la MT i-ésima, considerando el orden canónico introducido previamente (se emplea la visión de MT como calculadora de una función). Sea M, que a partir del input i hace:

1) Hacer n := 0.
2) Crear el siguiente string w, según el orden canónico.
3) Verificar si w es un código válido de MT. Si no, volver a (2).
4) Si i = n, imprimir w en la cinta de output y detenerse en q_A.
5) Hacer n := n + 1.
6) Volver a (2).

En (2) se generan strings w con símbolos que pueden formar parte del código de una MT, en el orden canónico (a partir de un ordenamiento alfabético establecido entre los símbolos). La factibilidad de (3) se probó en el lema anterior. Cuando en (4) se cumple la igualdad i = n, significa que se obtuvo el código $<M_i>$.

□

De esta manera se obtiene una enumeración de todas las MT, según el ordenamiento de sus codificaciones en el orden canónico: M_0, M_1, M_2, … Notar que puede darse, para índices i, k, tales que $i \neq k$, que $L(M_i) = L(M_k)$.

En la prueba del segundo lema, la MT M computa una función f que está definida para todo número natural. Si M_f indica que M computa la función f, se define:

Definición. Una función f : A → B es *total computable* sii existe una MT M_f que a partir de cualquier a ε A, se detiene y computa f(a) en su cinta de output.

□

Las funciones totales computables se utilizarán frecuentemente en el capítulo siguiente, en que se introducen las reducciones de problemas. Ahora se está en condiciones de probar el Teorema del Problema de la Detención, es decir que HP ε (RE – R).

Prueba del teorema. Se va a probar: (a) HP ε RE, y (b) HP ∉ R.

Prueba de la parte (a). HP ε RE:

Sea la siguiente MT M_{HP}, que a partir del input v hace:

1) Si v no es un código válido (<M>, w), rechazar.
2) Simular M a partir de w. Si M se detiene, aceptar.

La figura 1.3.1 ilustra el comportamiento de la MT M_{HP} .

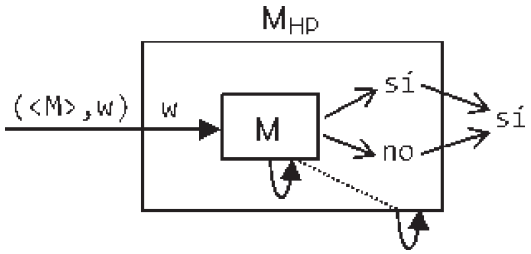


Figura 1.3.1

Se cumple que $L(M_{HP})$ = HP: v ε $L(M_{HP})$ ↔ M_{HP} acepta v ↔ v = (<M>, w) y M se detiene a partir de w ↔ v ε HP.

Prueba de la parte (b). HP $\notin$ R:

La prueba se basa en la técnica de *diagonalización*. Sea la matriz T de la figura 1.3.2:

T	w_0	w_1	w_2	- - -	w_i	- - -
M_0	t_{00}	t_{01}	t_{02}	- - -	t_{0i}	- - -
M_1	t_{10}	t_{11}	t_{12}	- - -	t_{1i}	- - -
M_2	t_{20}	t_{21}	t_{22}	- - -	t_{2i}	- - -
- - -	- - -	- - -	- - -	- - -	- - -	- - -
M_i	t_{i0}	t_{i1}	t_{i2}	- - -	t_{ii}	- - -
- - -	- - -	- - -	- - -	- - -	- - -	- - -

Figura 1.3.2

Las MT M y los strings w se disponen según el orden canónico. T es una matriz infinita de unos y ceros que se interpreta de la siguiente manera: $T[M_i, w_k] = t_{i\,k} = 1$ ó 0, según M_i se detiene o no a partir de w_k, respectivamente.
Sea el lenguaje $D = \{w_i \mid M_i$ se detiene a partir de $w_i\}$. Es decir que los elementos de D son los strings w_i tales que $T[M_i, w_i] = 1$.
Se va a probar: (b1) HP ε R $\rightarrow$ D ε R, y (b2) D $\notin$ R, por lo que se cumplirá que HP $\notin$ R.

Prueba de la parte (b1). HP ε R $\rightarrow$ entonces D ε R:

Si el lenguaje HP fuera recursivo también lo sería el lenguaje D, dado que D es un caso particular de HP, que en lugar de considerar todos los pares $(<M_i>, w_k)$ sólo considera los pares $(<M_i>, w_i)$. Formalmente:

Si HP ε R, entonces existe una MT M_{HP} que se detiene siempre y reconoce HP. Para probar que D ε R se va a construir una MT M_D que se detiene siempre y reconoce D. A partir del input w, M_D hace:

1) Encontrar el índice i tal que $w = w_i$, según el orden canónico.
2) Generar $<M_i>$.
3) Simular M_{HP} a partir del input $(<M_i>, w_i)$, y aceptar (respectivamente rechazar) si M_{HP} acepta (respectivamente rechaza).

En (1), la MT M_D genera canónicamente todos los strings w, hasta que encuentra el i-ésimo. El paso (2) ya se explicó en la prueba del Lema 2 anterior. Notar en (3) que un fragmento de la función de transición de M_D será la función de transición de M_{HP}, la cual a su vez simula M_i a partir de w_i.

Como M_{HP} se detiene siempre, entonces M_D se detiene siempre.

Además, $D = L(M_D)$: $w_i \varepsilon D \leftrightarrow M_i$ se detiene a partir de $w_i \leftrightarrow M_{HP}$ acepta $(<M_i>, w_i) \leftrightarrow M_D$ acepta w_i.

Prueba de la parte (b2). $D \notin R$:

Supóngase que $D \varepsilon R$. Sea M_D una MT que se detiene siempre y reconoce el lenguaje D. A partir de M_D se va a construir la siguiente MT M_C, que dado el input w hace:

1) Simular M_D a partir de w.
2) Si M_D acepta, «entrar en loop». Y si M_D rechaza, aceptar (también puede rechazar).

La figura 1.3.3 ilustra el comportamiento de la MT M_C.

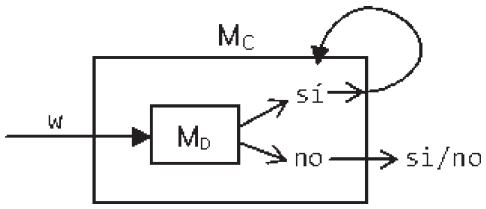


Figura 1.3.3

Queda como ejercicio para el lector, describir el fragmento de la función de transición de M_C que la hace «entrar en loop». En (2), da lo mismo que M_C acepte o rechace, si M_D rechaza, lo importante es que se detenga (la idea es que M_C «le lleve la contra» a M_D, como se verá enseguida).

Volviendo a la matriz T de la figura 1.3.2, dado que en ella están enumeradas todas las MT, la MT M_C es alguna MT M_m. Entonces, dada la MT M_D y el string w_m:

a) M_D acepta $w_m \rightarrow M_C = M_m$ no se detiene a partir de w_m, por la construcción de $M_C \rightarrow M_D$ rechaza w_m, por la definición de M_D (absurdo).

b) M_D rechaza $w_m \rightarrow M_C = M_m$ se detiene a partir de w_m, por la construcción de $M_C \rightarrow M_D$ acepta w_m, por la definición de M_D (absurdo).

Por lo tanto, no puede existir la MT M_C, y como ésta se construyó a partir de la MT M_D, entonces tampoco puede existir M_D, por lo que $D \notin R$.

□

En términos de la matriz T ha sucedido lo siguiente. La MT M_C tiene que ser alguna MT M_m de la enumeración de MT de T. La diagonal y la fila m de T coinciden en t_{mm}. Pero por cómo se construyó M_C a partir de M_D, se cumple que $t_{mm} = 1$ sii $t_{mm} = 0$, por lo que M_C no puede existir (y por lo tanto tampoco M_D).
De esta manera, se ha probado que hay lenguajes recursivamente numerables que no son recursivos. El lenguaje HP es uno de ellos, es computable pero no decidible. Se dirá que el problema de la detención no es decidible (porque el lenguaje que lo representa, es decir HP, no lo es).
Notar que aunque no existe una MT M_1 que resuelve en general el problema de la detención, sí en cambio existe una MT M_2 que lo resuelve en particular, para una determinada MT M_i y un determinado input w_k: M_2 responde «sí», o responde «no», porque M_i o bien se detiene o bien no se detiene a partir de w_k. Los problemas con una sola instancia son trivialmente decidibles y no serán considerados. En el próximo capítulo se tratará, a través de numerosos ejemplos, la existencia o inexistencia de soluciones para problemas con infinitas instancias.
Que los lenguajes recursivamente numerables no son todos los lenguajes, es decir que $RE \subset \mathcal{L}$, ya se probó en el capítulo anterior de dos maneras (asumiendo $R \subset RE$). Por un lado se demostró que si un lenguaje $L \varepsilon (RE - R)$, entonces $L^C \notin RE$, dado que se cumple $R = RE \cap$ CO-RE (por ejemplo, $HP^C \notin RE$). Por otro lado se encontró un lenguaje $L_{01} \notin (RE \cup$ CO-RE).
Una tercera forma de probar que $RE \subset \mathcal{L}$ se basa en la cardinalidad de los conjuntos infinitos:

- No puede haber más MT, y así lenguajes recursivamente numerables, que $|\Sigma^*|$.
- La cantidad de lenguajes de $\mathcal{L}$ es $|\mathcal{P}(\Sigma^*)|$.
- Se cumple que $|\Sigma^*| < |\mathcal{P}(\Sigma^*)|$, y por lo tanto hay más lenguajes que lenguajes recursivamente numerables.

Antes de pasar al siguiente capítulo, se presenta a continuación otro ejemplo de utilización de la técnica de diagonalización, en que se prueba de una cuarta manera que $RE \subset \mathcal{L}$.

Ejemplo. Se prueba por diagonalización que $RE \subset \mathcal{L}$.

Prueba. Sea la matriz T de la figura 1.3.4:

T	**w_0**	**w_1**	**w_2**	**- - -**	**w_i**	**- - -**
M_0	**t_{00}**	t_{01}	t_{02}	- - -	t_{0i}	- - -
M_1	t_{10}	**t_{11}**	t_{12}	- - -	t_{1i}	- - -
M_2	t_{20}	t_{21}	**t_{22}**	- - -	t_{2i}	- - -
- - -	- - -	- - -	- - -	**- - -**	- - -	- - -
M_i	t_{i0}	t_{i1}	t_{i2}	- - -	**t_{ii}**	- - -
- - -	- - -	- - -	- - -	- - -	- - -	**- - -**

Figura 1.3.4

Las MT M y los strings w aparecen en el orden canónico. T es una matriz infinita de unos y ceros que se interpreta de la siguiente manera: $T[M_i, w_k] = t_{i\,k} = 1$ ó 0, según M_i acepta o rechaza w_k, respectivamente. Por lo tanto, $L(M_i) = L_i = \{w_k \mid T[M_i, w_k] = 1\}$.

Sea $D = \{w_i \mid T[M_i, w_i] = 1\}$. Se cumple que $D^C \notin RE$:

$D^C \neq L_0$ porque $w_0 \in D^C \leftrightarrow w_0 \notin L_0$,
$D^C \neq L_1$ porque $w_1 \in D^C \leftrightarrow w_1 \notin L_1$,
$D^C \neq L_2$ porque $w_2 \in D^C \leftrightarrow w_2 \notin L_2$,
y así sucesivamente.
Es decir, D^C no es ninguno de los lenguajes L_i, y como los lenguajes L_i son todos los lenguajes recursivamente numerables, dado que la matriz T enumera todas las MT, entonces $D^C \notin RE$.

□

CAPÍTULO 1.4. REDUCCIONES DE PROBLEMAS

En este último capítulo de la primera parte del libro se introduce la técnica de reducción de problemas. Las reducciones permiten caracterizar problemas de una manera en general sencilla, a partir del conocimiento que se tiene de otros problemas. Esta técnica va a ser fundamental para el estudio de la complejidad computacional en la segunda parte del libro.
En la Sección 1.4.1 se definen las reducciones de problemas. En la Sección 1.4.2 se desarrollan varios ejemplos de reducciones, mediante las cuales se prueba la indecidibilidad de importantes problemas de la ciencia de la computación.
Como complemento, en la Sección 1.4.3 se presenta el Teorema de Rice, que se demuestra por una reducción de problemas. Utilizando este teorema se puede comprobar «sintácticamente» que un lenguaje, representante de un subconjunto propio del conjunto de los lenguajes recursivamente numerables, no es recursivo.

1.4.1. Definición y utilidad de las reducciones de problemas

La noción de *reducción de problemas* es muy simple: para resolver un problema P_1, se lo relaciona de alguna manera con otro problema P_2 cuya solución se conoce.

Definición. Sean L_1 y L_2 dos lenguajes incluidos en Σ^*. Existe una reducción de L_1 a L_2 sii existe una función total computable $f : \Sigma^* \rightarrow \Sigma^*$ tal que:

$$\forall w \in \Sigma^*: w \in L_1 \leftrightarrow f(w) \in L_2.$$

La función f se denomina *función de reducción*.

□

La figura 1.4.1 ilustra la definición de reducción de problemas. Que haya una reducción de L_1 a L_2 significa, entonces, que existe una MT que transforma todo string de L_1 en un string de L_2, y todo string no perteneciente a L_1 en un string no perteneciente a L_2.

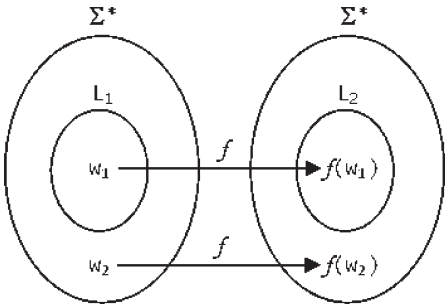


Figura 1.4.1

Si los lenguajes L_1 y L_2 representan, respectivamente, los problemas P_1 y P_2, será lo mismo referirse a una reducción de L_1 a L_2 que a una reducción de P_1 a P_2. La expresión

$$L_1 \alpha L_2,$$

denota que existe una reducción de L_1 a L_2. A veces, para simplificar la escritura, se identificará con M_f a una MT que computa la función de reducción f, y correspondientemente f_M será la función de reducción f computada por la MT M.

Ejemplo. Sean los lenguajes:

$L_{100} = \{(x, y, z) \mid x^{100} + y^{100} = z^{100}\}$, y
$L_{200} = \{(x, y, z) \mid x^{200} + y^{200} = z^{200}\}$.

Se cumple que $L_{200} \alpha L_{100}$. Sea la función de reducción $f : \Sigma^* \rightarrow \Sigma^*$, con $f((x, y, z)) = (x^2, y^2, z^2)$:

La función f es una función total computable: se puede construir fácilmente una MT M_f que, dada una terna (x, y, z), compute la terna (x^2, y^2, z^2).

$(x, y, z) \varepsilon L_{200} \leftrightarrow f(x, y, z) = (x^2, y^2, z^2) \varepsilon L_{100}$: $(x, y, z) \varepsilon L_{200} \leftrightarrow x^{200} + y^{200} = z^{200} \leftrightarrow (x^2)^{100} + (y^2)^{100} = (z^2)^{100} \leftrightarrow (x^2, y^2, z^2) \varepsilon L_{100}$.

Tal como lo refleja este ejemplo, para probar que $L_1 \alpha L_2$, hay que probar que: (a) existe una MT M_f que computa una función total f, y (b) $w \varepsilon L_1 \leftrightarrow f(w) \varepsilon L_2$, para todo string w de Σ^*.

Se cumple que si $L_1 \alpha L_2$ y existe una MT que reconoce L_2, entonces existe una MT que reconoce L_1, lo que se prueba a continuación. De esto surge la denominación de reducción, porque se puede obtener la solución de un problema nuevo a partir de la solución de un problema conocido.

Teorema. Si $L_1 \alpha L_2$, entonces $L_2 \varepsilon R \rightarrow L_1 \varepsilon R$, y $L_2 \varepsilon RE \rightarrow L_1 \varepsilon RE$.

Prueba. Se probarán por separado los dos casos.

Se cumple que si $L_1 \alpha L_2$, y $L_2 \varepsilon R$, entonces $L_1 \varepsilon R$:

Idea general: componiendo la MT M_f que computa la función de reducción f, con la MT M_2 que se detiene siempre y reconoce el lenguaje L_2, se obtiene una MT M_1 que se detiene siempre y reconoce el lenguaje L_1 (ver la figura 1.4.2).

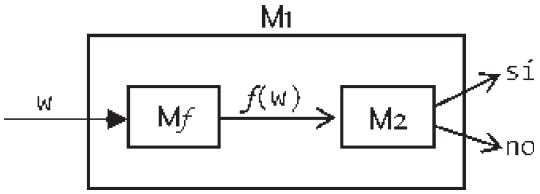


Figura 1.4.2

Formalmente: dadas las hipótesis, sea M_f la MT que computa la función de reducción f, con $w \varepsilon L_1 \leftrightarrow f(w) \varepsilon L_2$, y sea M_2 la MT que se detiene siempre y reconoce L_2. La MT M_1 que trabaja de la siguiente manera se detiene siempre y reconoce L_1 (por lo que se cumplirá que $L_1 \varepsilon R$):

1) Simular M_f a partir del input w, y obtener f(w).
2) Simular M_2 a partir de f(w), y aceptar sii M_2 acepta.

La MT M_1 se detiene siempre porque M_f y M_2 se detienen siempre.

Además, $L_1 = L(M_1)$:

$w \varepsilon L_1 \rightarrow M_f$ a partir de w computa $f(w) \varepsilon L_2 \rightarrow M_2$ a partir de f(w) se detiene en su estado $q_A \rightarrow M_1$ a partir de w se detiene en su estado $q_A \rightarrow w \varepsilon L(M_1)$.

$w \notin L_1 \rightarrow M_f$ a partir de w computa $f(w) \notin L_2 \rightarrow M_2$ a partir de f(w) se detiene en su estado $q_R \rightarrow M_1$ a partir de w se detiene en su estado $q_R \rightarrow w \notin L(M_1)$.

Por otro lado, se cumple que si $L_1 \alpha L_2$, y $L_2 \varepsilon RE$, entonces $L_1 \varepsilon RE$:

La idea general es como la de la parte (a), salvo que ahora la MT M_2 que reconoce L_2 puede no detenerse. Formalmente, si M_f es la MT que computa la función de reducción f, con $w \varepsilon L_1 \leftrightarrow f(w) \varepsilon L_2$, y M_2 es la MT que reconoce L_2, la siguiente MT M_1 reconoce L_1 (por lo que se cumplirá que $L_1 \varepsilon RE$):

1) Simular M_f a partir del input w, y obtener f(w).
2) Simular M_2 a partir de f(w), y aceptar sii M_2 acepta.

Se cumple que $L_1 = L(M_1)$:

$w \varepsilon L_1 \rightarrow M_f$ a partir de w computa $f(w) \varepsilon L_2 \rightarrow M_2$ a partir de f(w) se detiene en su estado $q_A \rightarrow M_1$ a partir de w se detiene en su estado $q_A \rightarrow w \varepsilon L(M_1)$.

$w \notin L_1 \rightarrow M_f$ a partir de w computa $f(w) \notin L_2 \rightarrow M_2$ a partir de f(w) se detiene en su estado q_R o no se detiene $\rightarrow M_1$ a partir de w se detiene en su estado q_R o no se detiene $\rightarrow w \notin L(M_1)$.

Corolario. Si $L_1 \alpha L_2$, entonces: (a) $L_1 \notin R \rightarrow L_2 \notin R$, y (b) $L_1 \notin RE \rightarrow L_2 \notin RE$.

□

El corolario establece que las reducciones se pueden emplear también para probar que un lenguaje L no es recursivo o no es recursivamente

numerable (en realidad, sirven para probar la no pertenencia a una gama mucho mayor de clases de lenguajes, como se verá en la segunda parte del libro). Más aún, las reducciones son más útiles para las pruebas «negativas» (las de no pertenencia) que para las «positivas» (las de pertenencia), las cuales se resuelven en general por medio de la construcción de MT.
Por otro lado, en comparación con la técnica de diagonalización, el empleo de la reducción de problemas es en general más sencillo. Las pruebas basadas en la diagonalización suelen ser dificultosas (así se probó que el lenguaje HP $\notin$ R). En cambio, las reducciones permiten «poblar» las distintas clases de lenguajes del mapa de la computabilidad y decidibilidad de una manera más simple, como se observará en los ejemplos de la siguiente sección.

1.4.2. Ejemplos de pruebas por reducciones de problemas

El primer ejemplo de reducción que se va a presentar en esta sección, se relaciona con un problema clásico como el de la detención, que es el problema de la aceptación (o pertenencia). Se va a probar que este problema no es decidible, a partir de que el problema de la detención tampoco lo es.

Ejemplo. Sea L_U = {(<M>, w) | M acepta w}. Se cumple que $L_U \notin$ R.

El lenguaje L_U se conoce también como el *lenguaje universal*, y representa el problema de la aceptación, que se enuncia de la siguiente manera: «Dada la MT M y el input w, ¿acaso M acepta w?». Se prueba fácilmente, lo que queda como ejercicio para el lector, que el lenguaje L_U es recursivamente numerable.

Prueba. Se hará: HP α L_U. Como HP $\notin$ R, entonces $L_U \notin$ R, por el corolario presentado al final de la sección anterior. Así se probará que el problema de la aceptación es indecidible.
Se propone una función de reducción f tal que f((<M>, w)) = (<M'>, w), donde M' es una MT muy similar a la MT M: se comporta como M, salvo que cuando M se detiene, ya sea en el estado q_A como en el estado q_R, M' siempre acepta.

La figura 1.4.3 ilustra la reducción planteada.

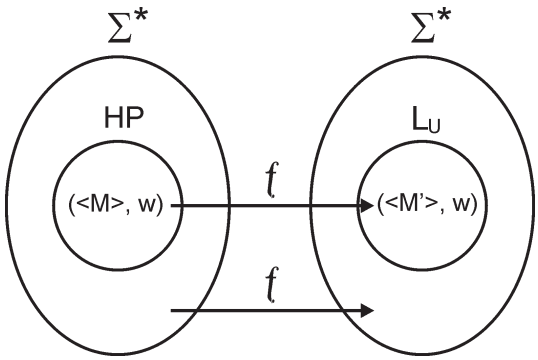


Figura 1.4.3

La función f es una función total computable:

Existe una MT M_f que genera adecuadamente pares (<M'>, w) a partir de pares (<M>, w). Dado el input v, M_f primero chequea si v es un string válido (<M>, w), y si no lo es genera como output el string «1», que no es un par válido (<M'>, w), de modo tal que de strings inválidos fuera de HP la MT M_f pasa a strings inválidos fuera de L_U. En caso contrario, para generar el output, la MT M_f modifica las 5-tuplas de <M> reemplazando todo estado q_R por el estado q_A.

Se cumple además que (<M>, w) ε HP $\leftrightarrow$ f((<M>, w)) = (<M'>, w) ε L_U:

(<M>, w) ε HP $\leftrightarrow$ M se detiene a partir de w $\leftrightarrow$ M' acepta w $\leftrightarrow$ (<M'>, w) ε L_U.

□

Para probar que L_U no es recursivo, se definió una reducción desde un lenguaje no recursivo (el lenguaje HP) al lenguaje L_U. Este es el esquema de prueba derivado del corolario que se presentó en la sección anterior. L_U no puede ser recursivo, porque si lo fuera también lo sería HP (absurdo): combinando M_f con la supuesta MT que acepta L_U y se detiene siempre, se obtendría una MT que acepta HP y se detiene siempre.

Planteando lo anterior de una forma más general: probando que existe una reducción de L_1 a L_2, se demuestra que el lenguaje L_2 es tan o más difícil que el lenguaje L_1, desde el punto de vista de la computabilidad y decidibilidad. Lo mismo sucederá en el marco de la complejidad computacional, utilizando reducciones con determinado costo en términos de tiempo o espacio.
Volviendo al ejemplo, no había muchas opciones de lenguajes conocidos para elegir, de los cuales reducir al lenguaje nuevo L_U. Con un conjunto más rico de alternativas, que se irán presentando en los ejemplos siguientes, la elección del lenguaje conocido podrá basarse en determinadas características del mismo y del lenguaje nuevo, para facilitar la construcción de M_f.
Notar que si se hubiera pretendido determinar si L_U es o no un lenguaje recursivo, una primera aproximación de demostración podría haber sido la siguiente:

a) Construir una MT que reciba como input un par (<M>, w) y simule M a partir de w.
b) Como M puede no detenerse a partir de w, entonces determinar que L_U no es recursivo.

Obviamente esta prueba no es correcta, pero de todos modos un primer intento de esta naturaleza puede servir para orientar la demostración hacia una prueba «positiva» (la construcción de una MT que pruebe que $L_U \in R$), o «negativa» (una reducción que pruebe que $L_U \notin R$), que fue finalmente la que se llevó a cabo. Esta aproximación se presenta en algunos ejemplos, como el siguiente.

Ejemplo. Sea el problema: «¿Acaso la MT M acepta todos los inputs?». Se pretende determinar si dicho problema es decidible o indecidible.

El lenguaje que representa el problema es $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\}$. Por lo tanto, hay que determinar si L_{Σ^*} es o no un lenguaje recursivo.
Intuitivamente, no parece siquiera que el lenguaje L_{Σ^*} sea recursivamente numerable. Si $L_{\Sigma^*} \in RE$, entonces existe una MT M_{Σ^*} que se detiene en q_A sii su input <M> es tal que $L(M) = \Sigma^*$. Por lo tanto, M_{Σ^*} debe responder «sí», en tiempo finito, después de comprobar que M acepta los infinitos strings de Σ^*, lo que no parece razonable. Obviamente esto no es una demostración formal, pero permite orientar la prueba hacia la búsqueda de una reducción para probar que L_{Σ^*} no es recursivo.

Prueba. Se hará: $L_U \alpha L_{\Sigma^*}$. Como $L_U \notin R$ entonces $L_{\Sigma^*} \notin R$ (si fuera $L_{\Sigma^*} \varepsilon R$, entonces sería $L_U \varepsilon R$). De esta manera se probará que el problema de determinar si una MT acepta todos los inputs es indecidible.
Se propone una función de reducción f tal que f((<M>, w)) = $<M_w>$, donde M_w es una MT que: (a) borra su input v y lo reemplaza por el string w, (b) simula M a partir de w, y (c) acepta sii M acepta.
Se comprueba fácilmente que si M acepta w, entonces $L(M_w) = \Sigma^*$, y si M no acepta w, entonces $L(M_w) = \varnothing$ (el lenguaje $L(M_w)$ es distinto de Σ^*, que es lo que se necesita).

La figura 1.4.4 ilustra la reducción planteada.

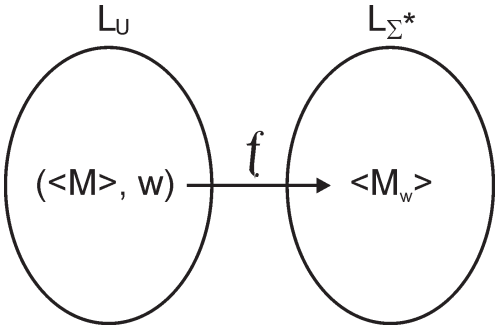


Figura 1.4.4

La función f es una función total computable:

Existe una MT M_f que genera adecuadamente $<M_w>$ a partir de (<M>, w). M_f primero chequea si el input (<M>, w) es válido, y si no lo es, genera como output el string «1», que no es un código válido $<M_w>$, de modo tal que de strings inválidos fuera de L_U, la MT M_f pasa a strings inválidos fuera de L_{Σ^*}. En caso contrario, para generar el output, la MT M_f le agrega al código <M> un fragmento inicial que borra el input y lo reemplaza por w.

Se cumple además que (<M>, w) $\varepsilon L_U \leftrightarrow$ f((<M>, w)) = $<M_w> \varepsilon L_{\Sigma^*}$:

(<M>, w) $\varepsilon L_U \leftrightarrow$ M acepta w $\leftrightarrow L(M_w) = \Sigma^* \leftrightarrow <M_w> \varepsilon L_{\Sigma^*}$.

□

En el ejemplo siguiente se demuestra que el lenguaje L_{Σ^*} no es siquiera recursivamente numerable, como se sugirió recién. Se va a construir una primera reducción de problemas para probar la no pertenencia de un lenguaje a la clase RE, lo que suele ser más difícil que las pruebas de no pertenencia a la clase R utilizando reducciones.

Ejemplo. Probar que $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\} \notin RE$.

Prueba. Se hará: $L_U{}^C \propto L_{\Sigma^*}$. Ya se probó previamente que el complemento L^C de un lenguaje L de (RE – R) no puede ser recursivamente numerable. Por lo tanto $L_U{}^C \notin RE$, y así, con la reducción propuesta se probará que $L_{\Sigma^*} \notin RE$ (si L_{Σ^*} fuera recursivamente numerable también lo sería el lenguaje $L_U{}^C$, por el corolario de la sección anterior).

Se propone una función de reducción f tal que $f((<M>, w)) = <M_w>$, donde M_w es una MT que, a partir del input v, hace:

1) Si (<M>, w) no es válido, entonces acepta v.
2) Si en cambio (<M>, w) es válido, entonces simula M a partir de w, a lo sumo |v| pasos (la MT M se puede detener antes), aceptando v sii M no acepta w.

Se comprueba fácilmente que:

a) Si M no acepta w, entonces $L(M_w) = \Sigma^*$ (por convención, cuando el string (<M>, w) es inválido se considera que M no acepta w).
b) Si M acepta w (en k pasos), entonces $L(M_w) = \{v \mid |v| < k\}$ (el lenguaje $L(M_w)$ es distinto de Σ^*, que es lo que se necesita).

La función f es una función total computable:

Existe una MT M_f que genera adecuadamente $<M_w>$ a partir de (<M>, w). M_f primero chequea si el input (<M>, w) es válido, y si no lo es genera como output el código $<M_w>$, tal que M_w es una MT que acepta todos sus inputs. En caso contrario, para generar el output, la MT M_f le agrega al código <M> un fragmento que calcula el tamaño i del input, decrementa i en 1 toda vez que se ejecuta un paso, detiene la ejecución cuando i = 0, y acepta sii no se alcanza al final el estado q_A.

Se cumple además que $(<M>, w) \varepsilon L_U{}^C \leftrightarrow f((<M>, w)) = <M_w> \varepsilon L_{\Sigma^*}$:

(<M>, w) ε L_U^C $\leftrightarrow$ M no acepta w $\leftrightarrow$ $L(M_w) = \Sigma^* \leftrightarrow <M_w> \varepsilon L_{\Sigma^*}$.

□

Otro problema clásico es el de la equivalencia de MT. Por medio de una reducción de problemas se va a probar que dicho problema no es decidible.

Ejemplo. Sea el problema: «¿Acaso la MT M_1 es equivalente a la MT M_2?». Se pretende determinar si dicho problema es decidible o indecidible.

El lenguaje que representa el problema es $L_{EQ} = \{(<M_1>, <M_2>) \mid L(M_1) = L(M_2)\}$. Hay que determinar si L_{EQ} es o no un lenguaje recursivo.

Intuitivamente, no parece siquiera que L_{EQ} ε RE. Si lo fuera, existiría una MT M_{EQ} que en tiempo finito responde «sí» cuando dado el input (<M_1>, <M_2>), M_{EQ} comprueba que las MT M_1 y M_2 aceptan y rechazan exactamente los mismos strings del conjunto infinito Σ^*, lo que no parece razonable. Con esta idea, se va a buscar una reducción para probar que L_{EQ} no es recursivo (en otras palabras, que el problema de la equivalencia de MT es indecidible). Se probará directamente que $L_{EQ} \notin$ RE.

Prueba. Se hará: L_{Σ^*} α L_{EQ}. Como $L_{\Sigma^*} \notin$ RE, entonces $L_{EQ} \notin$ RE (si fuera L_{EQ} ε RE, entonces también se cumpliría que L_{Σ^*} ε RE).

Se propone una función de reducción f tal que f(<M>) = (<M>, <M_{Σ^*}>), con $L(M_{\Sigma^*}) = \Sigma^*$.

La función f es una función total computable:

Existe una MT M_f que a partir del input <M> genera el output (<M>, <M_{Σ^*}>), siendo <M_{Σ^*}> el código de una MT que acepta todos los inputs.

Se cumple además que <M> ε L_{Σ^*} $\leftrightarrow$ f(<M>) = (<M>, <M_{Σ^*}>) ε L_{EQ}:

<M> ε L_{Σ^*} $\leftrightarrow$ $L(M) = \Sigma^* \leftrightarrow L(M) = L(M_{\Sigma^*})$ $\leftrightarrow$ (<M>, <M_{Σ^*}>) ε L_{EQ} (por convención, cuando el código <M> es inválido se considera que $L(M) = \varnothing$).

□

Como último ejemplo de prueba por reducción de problemas, se va a demostrar a continuación que no se puede decidir si: (a) el lenguaje aceptado por una MT es recursivo, y (b) el lenguaje aceptado por una MT no es recursivo.

Ejemplo. Primero se va a probar la parte (a), es decir, que es indecidible determinar si, dada la MT M, L(M) es un lenguaje recursivo. El lenguaje que representa el problema es:

L_{REC} = {<M> | L(M) ε R}. Se va a probar que $L_{REC} \notin$ RE.

Prueba parte (a). Se hará: $L_U{}^C \alpha L_{REC}$. Como $L_U{}^C \notin$ RE, entonces se cumplirá que $L_{REC} \notin$ RE.

Se propone una función de reducción f tal que f((<M>, w)) = $<M_w>$, donde M_w es una MT que, a partir del input v hace:

1) Simula M a partir de w. Si M no acepta, entonces M_w no acepta.
2) Simula M_U, que es una MT que reconoce el lenguaje universal L_U, a partir del input v, que si es un string válido en este caso tendrá la forma (<M'>, w'). M_w acepta sii M_U acepta (por convención, si (<M'>, w') es inválido, entonces la MT M_U lo rechaza).

Se comprueba fácilmente que si M no acepta w, entonces $L(M_w) = \varnothing$ (que es un lenguaje recursivo), y si en cambio M acepta w, entonces $L(M_w) = L_U$ (que no es un lenguaje recursivo).

La función f es una función total computable:

Existe una MT M_f que genera adecuadamente el output $<M_w>$ a partir del input (<M>, w). La MT M_f primero chequea si (<M>, w) es válido, y si no lo es genera un código $<M_\varnothing>$, con $L(<M_\varnothing>) = \varnothing$, de modo tal que de strings inválidos que por convención están dentro de $L_U{}^C$, M_f pasa a códigos $<M_w>$ de L_{REC}. En caso contrario, M_f genera $<M_w>$ fundamentalmente agregando a <M>, por un lado el código $<M_U>$, y por otro lado un fragmento que reemplaza al input v por w y luego restituye a v como input.

Además se cumple que (<M>, w) ε $L_U{}^C \leftrightarrow$ f((<M>, w)) = $<M_w> \varepsilon L_{REC}$:

(<M>, w) ε $L_U{}^C \leftrightarrow$ M no acepta w $\leftrightarrow L(M_w) = \varnothing \leftrightarrow <M_w> \varepsilon L_{REC}$.

□

Ahora se va a probar la parte (b), es decir, que es indecidible determinar si, dada la MT M, L(M) es un lenguaje no recursivo. El lenguaje que representa el problema es:

$L_{REC}{}^{C}$ = {<M> | L(M) $\notin$ R}. Se va a probar que también $L_{REC}{}^{C} \notin$ RE.

Prueba parte (b). Se hará: $L_U{}^{C} \propto L_{REC}{}^{C}$. Como $L_U{}^{C} \notin$ RE, entonces valdrá $L_{REC}{}^{C} \notin$ RE.

Se propone una función de reducción f tal que f((<M>, w)) = $<M_w>$, donde M_w es una MT que, a partir del input v, simula en paralelo M a partir de w y M_U a partir de v, aceptando sii alguna de las dos MT aceptan. Se comprueba fácilmente que si M no acepta w, entonces $L(M_w) = L_U$ (que no es recursivo), y si M acepta w, entonces $L(M_w) = \Sigma^*$ (que es recursivo).

La función f es una función total computable:

Existe una MT M_f que genera adecuadamente $<M_w>$ a partir de (<M>, w). M_f primero chequea si (<M>, w) es válido, y si no lo es genera el código $<M_U>$, de modo tal que de strings inválidos que por convención están dentro de $L_U{}^{C}$, la MT M_f pasa a códigos $<M_w>$ de $L_{REC}{}^{C}$. En caso contrario, M_f genera un output $<M_w>$ fundamentalmente agregándole a <M>, el código $<M_U>$ más un fragmento que permite ejecutar en paralelo M a partir de w y M_U a partir de v.

Se cumple además que (<M>, w) ε $L_U{}^{C} \leftrightarrow$ f((<M>, w)) = $<M_w> \varepsilon L_{REC}{}^{C}$:

(<M>, w) ε $L_U{}^{C} \leftrightarrow$ M no acepta w $\leftrightarrow$ $L(M_w) = L_U \leftrightarrow <M_w> \varepsilon L_{REC}{}^{C}$.

□

Los ejemplos anteriores reflejan la utilidad de la técnica de reducción de problemas, que se empleará también para categorizar problemas en el marco de la complejidad computacional.
Diagonalizando se encontró un primer lenguaje no recursivo, el lenguaje HP, y a partir de él, por medio de reducciones de problemas, se probó la indecidibilidad de importantes problemas de la teoría de lenguajes formales.
Las reducciones construidas se han basado en distintos modelos o «recetas», como por ejemplo:

– La generación de $<M_w>$ a partir de (<M>, w), tal que M_w ignora sus inputs v y simula directamente M a partir de w.

– Una variante del modelo anterior, donde M_w simula primero M a partir de w, y después «filtra» simulando otra MT M' a partir de v. La elección de M' se relaciona con el lenguaje al que se pretende reducir.
– Otra variante del primer modelo, en que M_w simula M a partir de w teniendo en cuenta alguna característica de sus inputs v (por ejemplo, ejecutando sólo |v| pasos de M).

Estas y otras «recetas» se relacionan con los lenguajes que intervienen en las reducciones (en la parte de complejidad computacional se describen algunos ejemplos más). En particular, el Teorema de Rice, que se presenta a continuación y que se prueba mediante una reducción de problemas, permite demostrar de una manera muy sencilla que un determinado tipo de lenguajes no son recursivos.

1.4.3. Teorema de Rice

El Teorema de Rice establece que es indecidible determinar si el lenguaje L aceptado por una MT M tiene una determinada propiedad, salvo la propiedad trivial de ser recursivamente numerable. Por ejemplo, es indecidible determinar si L(M) es finito, infinito, $\varnothing$, Σ^*, recursivo, no recursivo, etc. Más precisamente:

Teorema de Rice. Si $\vartheta = \{L_1, L_2, \ldots, L_i, \ldots\}$ cumple que $\varnothing \subset \vartheta \subset RE$, entonces el lenguaje de códigos de MT asociados a ϑ, es decir $L_\vartheta = \{<M_1>, <M_2>, \ldots, <M_i>, \ldots\}$ tal que $\forall i$: $L(M_i) = L_i$, no es un lenguaje recursivo.

El conjunto ϑ representa una *propiedad no trivial* de RE. Intuitivamente, es razonable que el lenguaje L_ϑ no sea recursivo: probar que $L(M_i)$ cumple una determinada propiedad, requiere considerar los infinitos strings de Σ^*. El teorema no establece nada con respecto a la pertenencia de L_ϑ a la clase RE, porque en este caso se permite la no detención de una MT para rechazar el input. Al final de este capítulo se muestra un ejemplo de conjunto ϑ, cuyo lenguaje asociado L_ϑ se prueba que es recursivamente numerable.
El teorema tampoco establece nada cuando los códigos <M> no se definen en términos de los lenguajes que las MT M reconocen, sino que se relacionan directamente con características de las MT (por ejemplo,

MT con un número par de estados, MT que no recorren más de K celdas distintas, etc). Al final del capítulo se presenta un ejemplo de lenguaje de códigos de MT de este tipo, y se prueba que es recursivo.
La prueba del Teorema de Rice es un ejemplo más de reducción de problemas desde el lenguaje universal L_U, y se desarrolla a continuación.

Prueba. Sea $\varnothing \subset \vartheta \subset RE$, y $L_\vartheta = \{<M> \mid L(M) \in \vartheta\}$. La prueba se divide en dos casos.

Prueba del caso (a), en que $\varnothing \notin \vartheta$:

Se va a probar que: $L_U \alpha L_\vartheta$. Como $L_U \notin R$, entonces será $L_\vartheta \notin R$.

Se propone una función de reducción f tal que f((<M>, w)) = <M'>. Sea $L_1 \neq \varnothing \in \vartheta$ (L_1 existe porque $\vartheta \neq \varnothing$ y $\varnothing \notin \vartheta$), y sea M_1 la MT que reconoce L_1. La MT M', a partir del input v, hace:

1) Simula M a partir de w, y si M no acepta, entonces no acepta.
2) Simula M_1 a partir de v, y acepta sii M_1 acepta.

Se comprueba fácilmente que si M acepta w, entonces $L(M') = L_1 \in \vartheta$, y si M no acepta w, entonces $L(M') = \varnothing \notin \vartheta$.

La función f es una función total computable:

Existe una MT M_f que genera adecuadamente <M'> a partir de (<M>, w). M_f primero chequea si (<M>, w) es válido, y si no lo es genera como output el string «1» (en este caso vale por convención $L(M') = \varnothing$, con $\varnothing \notin \vartheta$ por hipótesis), de modo tal que de strings inválidos fuera de L_U la MT M_f pasa a strings inválidos fuera de L_ϑ. En caso contrario, M_f genera el output <M'> fundamentalmente agregándole a <M> el código $<M_1>$, más un fragmento relacionado con la simulación de M a partir de w y de M_1 a partir de v.

Se cumple además que $(<M>, w) \in L_U \leftrightarrow f((<M>, w)) = <M'> \in L$:

$(<M>, w) \in L_U \leftrightarrow$ M acepta w $\leftrightarrow L(M') = L_1 \in \vartheta \leftrightarrow <M'> \in L_\vartheta$.

Prueba del caso (b), en que $\varnothing \in \vartheta$:

Se cumple que $\varnothing \notin (RE - \vartheta) = \vartheta'$, con $\varnothing \subset \vartheta' \subset RE$. Entonces por la parte (a), vale $L_{\vartheta'} \notin R$. Pero $L_{\vartheta'} = L_\vartheta{}^C$, porque:

<M> ε L_ϑ, $\leftrightarrow$ L(M) ε ϑ' $\leftrightarrow$ L(M) $\notin$ ϑ $\leftrightarrow$ <M> $\notin$ L_ϑ $\leftrightarrow$ <M> ε L_ϑ^C (los códigos inválidos de MT se consideran por convención en L_ϑ, dado que $\varnothing$ ε ϑ).

Finalmente, como $L_\vartheta^C \notin R$, entonces también se cumple $L_\vartheta \notin R$, y por lo tanto otra vez se llegó a probar, a partir de ϑ, que el lenguaje asociado L_ϑ de códigos de MT no es recursivo.

□

Ejemplo. Sea L_{inf} = {<M> | L(M) es infinito}. Se cumple por la aplicación del Teorema de Rice que el lenguaje $L_{inf} \notin R$.

Prueba. El conjunto ϑ asociado al lenguaje L_{inf} es el subconjunto de RE de los lenguajes infinitos.

Se cumple que $\varnothing \subset \vartheta$ porque, por ejemplo, $\Sigma^* \varepsilon \vartheta$. Y se cumple que $\vartheta \subset$ RE porque, por ejemplo, $\varnothing \notin \vartheta$.

De esta manera se cumple que L_{inf} no es recursivo.

□

Los siguientes dos ejemplos cierran este último capítulo de la primera parte del libro. En el ejemplo siguiente se prueba que un determinado lenguaje de códigos de MT es recursivo. No se puede aplicar el Teorema de Rice, porque los códigos <M> no se definen en términos de los lenguajes L(M).

Ejemplo. Sea L_{20} = {<M> | M es una MTD con una cinta, y a partir del input vacío λ nunca sale de las celdas 1 a 20}. Por convención, la celda 1 es la apuntada inicialmente por el cabezal, y las celdas que le siguen a la derecha se denominan 2, 3, etc. Se pide determinar si el lenguaje L_{20} es recursivo.

No se puede aplicar el Teorema de Rice, porque <M> no se define en términos de L(M). Se va a probar, por construcción de una MT, que L_{20} ε R.

Prueba. Si una MTD M con una cinta no sale del intervalo de celdas 1 a 20, entonces existe un número máximo de configuraciones distintas C

por las que pasa M antes de «entrar en loop». Se cumple que C = 20.|Q|.|Γ|^{20}, porque el cabezal varía en el intervalo de celdas 1 a 20, los estados pertenecen a Q, y los símbolos de las celdas pertenecen a Γ.
Se va a construir una MT M_{20} que reconoce L_{20} y se detiene siempre, valiéndose de la cota C. La idea general es que si al cabo de C pasos, una MT M de las características mencionadas no se detiene, significa que «entró en loop».
Sea la siguiente MT M_{20} con 5 cintas, que a partir del input <M> en la cinta 1 hace:

1) Hacer i := 1 en la cinta 3, siendo i la posición del cabezal de M.
2) Hacer n := 0 en la cinta 4, siendo n el número de pasos ejecutados por M.
3) Calcular y escribir C en la cinta 5.
4) Simular el paso siguiente de M, con input λ, en la cinta 2.
5) Actualizar adecuadamente el valor de i en la cinta 3. Si i = 0 ó i = 21, detenerse en q_R. Si M se detuvo, detenerse en q_A.
6) Hacer n := n + 1 en la cinta 4. Si n = C, detenerse en q_A.
7) Volver a (4).

En (3), C se calcula a partir de los valores |Q| y |Γ| que se obtienen de <M>. En (5), el valor de i se incrementa en 1, se decrementa en 1 o se mantiene, según el movimiento de M sea R, L ó S, respectivamente.

La MT M_{20} se detiene siempre, porque alguna de las tres posibilidades de detención referenciadas en los pasos anteriores debe cumplirse, cualquiera sea <M>.

$L_{20} = L(M_{20})$: <M> ε L_{20} ↔ M, a partir de λ, no sale de las celdas 1 a 20 ↔ M_{20} se detiene, a partir de <M>, en su estado q_A ↔ <M> ε $L(M_{20})$.

□

El ejemplo anterior muestra que limitando el espacio de trabajo de una MT, se puede acotar su número de pasos, es decir su tiempo de trabajo, teniendo en cuenta la cantidad de configuraciones distintas. Esta relación entre el espacio y el tiempo consumidos por una MT, se estudia en la parte de complejidad computacional.
En el último ejemplo, que se presenta a continuación, se prueba que el lenguaje de códigos de MT que reconocen el lenguaje ∅ no es recursivamente numerable. Lo interesante de la prueba es que combina

la aplicación del Teorema de Rice con una prueba por construcción de una MT.

Ejemplo. Sea $L_{\varnothing} = \{<M> \mid L(M) = \varnothing\}$. Se prueba que $L_{\varnothing} \notin RE$.

Prueba. Queda como ejercicio para el lector, probar por la aplicación del Teorema de Rice que el lenguaje $L_{\varnothing}$ no es recursivo. Se va a probar que $L_{\varnothing}^{C} = \{<M> \mid L(M) \neq \varnothing\} \in RE$, y por lo tanto se cumplirá que $L_{\varnothing} \notin RE$, porque de lo contrario valdría $L_{\varnothing} \in R$. De esta manera, el lenguaje $L_{\varnothing}^{C}$ es un ejemplo de lenguaje de códigos de MT asociado a una propiedad no trivial de RE, que es recursivamente numerable.
Se va a construir una MT $M_{\varnothing}^{C}$ que reconoce $L_{\varnothing}^{C}$. La MT $M_{\varnothing}^{C}$, a partir del input w, hace:

1) Si w = <M> es inválido, rechazar.
2) Hacer i := 0.
3) Simular M exactamente i pasos, a partir de todos los strings v tales que $|v| \leq i$. Si en algún caso M acepta, aceptar.
4) Hacer i := i+1 y volver a (3).

$L_{\varnothing}^{C} = L(M_{\varnothing}^{C})$: $<M> \in L_{\varnothing}^{C} \leftrightarrow$ M acepta un string v $\leftrightarrow M_{\varnothing}^{C}$ detecta que M acepta un string v $\leftrightarrow M_{\varnothing}^{C}$ acepta <M> $\leftrightarrow <M> \in L(M_{\varnothing}^{C})$.

□

Apéndice 1

A1.1. Equivalencia entre máquinas de Turing y máquinas RAM

En este breve capítulo, se prueba la equivalencia entre las MT y las *máquinas RAM* (por *Random Access Machines*). Las RAM constituyen otro modelo de las computadoras, estructuralmente más similar, y por eso se presentan en este apéndice.
Hasta ahora se ha tratado solamente la equivalencia entre modelos de MT. Como en este caso se comparan las MT con las RAM, debe aclararse qué significa que estos dos modelos sean equivalentes. La idea es la misma, es decir, hay que demostrar que el poder computacional de los dos tipos de máquinas es el mismo.
Las RAM tienen un número infinito de palabras de memoria, numeradas 0, 1, 2, …, cada una de las cuales puede almacenar un número entero, y un número finito de registros aritméticos que también pueden almacenar enteros. Están provistas por un conjunto adecuado de instrucciones, tales como LOAD, STORE, ADD, etc., con la semántica conocida.
Por las características de las RAM, convendrá utilizar un modelo de MT que no se presentó previamente, en el que las cintas son infinitas sólo a la derecha, es decir que tienen una primera celda, la de más a la izquierda. En este caso, además de detenerse en la forma habitual, las MT lo hacen si estando posicionadas en la primera celda se mueven a la izquierda, por lo que rechazan el input. Queda como ejercicio para el lector, probar la equivalencia entre los modelos de MT con cintas infinitas a ambos lados y cintas infinitas a la derecha.

Teorema. El modelo de las MT con cintas infinitas a la derecha es equivalente al modelo de las RAM.

Prueba. Sólo se presenta la idea general. Primero se verá que, dada una MT M con una cinta infinita a la derecha, existe una RAM R equivalente a M:

- El registro R_1 de R contiene la posición del cabezal de M.
- El registro R_2 de R contiene el estado corriente de M.

– Sea $[R_1]$ el contenido apuntado por la dirección almacenada en R_1 . Para simplificar la escritura, en lugar de LOAD, STORE, ADD, etc., se van a utilizar instrucciones de mayor nivel de abstracción, que pueden ser implementadas por las mismas. Si por ejemplo en M se tiene $\delta(q, x) = (q', x', L)$, entonces en R hay un conjunto de instrucciones de la forma siguiente:

IF ((R_2 = q) AND ($[R_1]$ = x)) THEN BEGIN R_2 := q'; $[R_1]$:= x'; R_1 := R_1 – 1 END.

– El programa completo de R es una iteración, cuyo cuerpo es una selección múltiple que considera todos los pares (q, x) de M. El programa termina sii M se detiene.

Ahora se verá que dada una RAM R, existe una MT M equivalente. Dada R, se construirá M con K cintas infinitas a la derecha, que simulará R.

En la cinta 1 de la MT M, están primero las palabras de la RAM R con valores (las instrucciones y el input), y luego hay símbolos «B». Por ejemplo, usando notación binaria, el contenido de la cinta 1 empezaría así:

$$\#0*v_0\#1*v_1\#10*v_2\#11*v_3\#100*v_4\#\ldots\#i*v_i\#\ldots,$$

donde v_i es el contenido, en binario, de la palabra i-ésima, es decir la palabra de la dirección i.

Los distintos registros aritméticos de R están en las cintas 2, 3, 4, …, de M.

M tiene dos cintas más:

a) Una cinta para almacenar el *location counter* (o LC) de R, el cual contiene la dirección de la palabra de la que debe leerse la próxima instrucción. Al comienzo vale cero.
b) Una cinta para almacenar el *memory address register* (o MAR) de R, el cual contiene la dirección de la palabra de la que debe leerse el próximo valor.

Supóngase que los primeros bits de una instrucción representan su código (LOAD, STORE, ADD, etc.), y los bits restantes representan la dirección del operando referenciado en la instrucción. En lo que sigue, se presenta un ejemplo concreto para describir la simulación.

Si en un momento dado, la cinta de M correspondiente al LC tiene el valor i, se hace:

1) M recorre su cinta 1 desde el extremo izquierdo y busca el string #i*.
2) Si M encuentra antes un «B», significa que no hay instrucción en la palabra i, es decir que R terminó, por lo que M se detiene.
3) En caso contrario, M procesa los bits siguientes a #i* hasta el próximo símbolo «#» (es decir, procesa v_i). Supóngase, por ejemplo, que los primeros bits de v_i representan la instrucción «ADD TO REGISTER 2», y que los bits restantes representan el número k. Entonces, la MT M hace lo siguiente:

   – Suma 1 en la cinta del LC (actualización para la próxima instrucción).
   – Copia k en la cinta del MAR (dirección del valor a sumar al registro 2).
   – Busca el string #k* en la cinta 1 desde el extremo izquierdo.
   – Si #k* no aparece, no hace nada (se asume k = 0). En caso contrario, obtiene v_k del string #k*v_k# , y lo suma al contenido de la cinta que representa el registro 2.

Este ciclo se repite para la siguiente instrucción, con el nuevo valor de la cinta del LC, y así sucesivamente, hasta que M eventualmente se detiene.

□

A1.2. Máquinas de Turing generadoras de lenguajes

Además de las visiones de MT como reconocedoras de lenguajes y calculadoras de funciones, existe una tercera visión como generadoras de lenguajes, que se presenta en este capítulo. Se muestra también la relación entre las visiones de MT que generan y reconocen lenguajes. En el final, se introduce una representación alternativa de los lenguajes, las gramáticas, que se siguen considerando en el capítulo siguiente.
Las MT que generan lenguajes tienen una cinta de output de solo escritura, en la que el cabezal se mueve solamente a la derecha, y los strings se separan por el símbolo especial «#» (ver la figura A1.2.1).

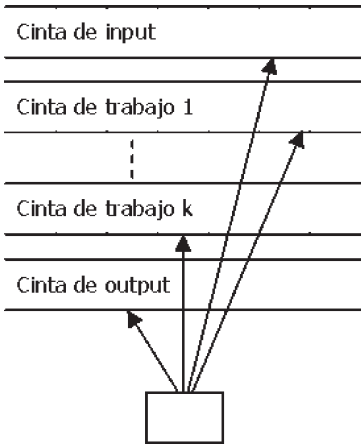


Figura A1.2.1

El lenguaje generado por una MT es el conjunto de strings que se imprimen en su cinta de output. Formalmente:

Definición. Una MT M *genera* el lenguaje L, sii todo string w ε L aparece al menos alguna vez en la cinta de output de M.

□

Los strings de la cinta de output se pueden repetir, y no aparecen en ningún orden determinado. Se asume que el input de la MT es el string vacío λ.
El lenguaje generado por M se denota con G(M). Así, utilizando las expresiones G(M) y L(M) puede identificarse qué visión de MT se está utilizando (generadora y reconocedora, respectivamente).
Se cumple que existe una MT M_1 que reconoce un lenguaje L sii existe una MT M_2 que lo genera, lo que se prueba a continuación:

Teorema. L ε RE ↔ ∃M: L = G(M).

Es decir, ya se sabe que un lenguaje L es recursivamente numerable sii existe una MT que lo reconoce. Ahora se probará que otra condición suficiente y necesaria es que exista una MT que lo genere (esto explica por qué se denomina lenguaje recursivamente numerable).

Prueba. Se probará primero que si L ε RE, entonces ∃M: L = G(M):

Sea la MT M_1, tal que $L(M_1) = L$. Se va a construir una MT M_2, tal que $G(M_2) = L$.

La MT M_2 genera, uno a uno, todos los strings w de Σ* en el orden canónico, y a partir de cada w simula M_1. Cuando M_1 acepta w, entonces M_2 lo imprime, seguido de «#», en la cinta de output. Hay que tener en cuenta que M_1 puede «entrar en loop» a partir de algunos w, por lo que las simulaciones deben llevarse a cabo de una manera particular. Formalmente, la MT M_2 hace:

1) i := 0.
2) Generar todos los strings w de Σ* de longitud a lo sumo i en el orden canónico.
3) Simular a lo sumo i pasos de M_1 a partir de cada w generado en (2).
4) Imprimir los strings w aceptados en (3) en la cinta de output, seguidos del símbolo «#».
5) i := i+1.
6) Volver a (2).

Notar que M_2 no se detiene aún si L es finito. Un mismo string se puede imprimir más de una vez (se puede evitar por medio de una cinta auxiliar). Además, el orden de impresión de los strings depende del comportamiento de M_1 (por eso no existe un orden determinado de impresión).

Se probará ahora que si ∃M: L = G(M), entonces L ε RE:

Sea la MT M_1, tal que $G(M_1) = L$. Se va a construir una MT M_2, tal que $L(M_2) = L$.

La MT M_2 tiene una cinta más que M_1, en la que está su input v. M_2 trabaja como M_1, pero toda vez que M_1 imprime un string w, M_2 lo compara con v, y si son iguales se detiene y acepta. Además, si M_1 en algún momento se detiene, entonces M_2 se detiene y rechaza.

□

Cuando los lenguajes son recursivos, se pueden listar sin repeticiones y en el orden canónico, como se prueba a continuación.

Teorema. L ε R ↔ ∃M: L = G(M) y M imprime los strings de L sin repeticiones y en el orden canónico.

Es decir, ya se sabe que un lenguaje L es recursivo sii existe una MT que lo reconoce y se detiene siempre. Ahora se probará que también es condición suficiente y necesaria que exista una MT que lo genere, sin repetir strings y en el orden canónico.

Prueba. Se probará primero que si L ε R, entonces ∃M: L = G(M) y M imprime L sin repetir strings y en el orden canónico:

Sea la MT M_1, tal que $L(M_1) = L$, y M_1 se detiene siempre. Se va a construir una MT M_2, tal que $G(M_2) = L$ y M_2 imprime L sin repeticiones y en el orden canónico.
Como se hizo en la prueba anterior, M_2 genera, uno a uno, todos los strings w de Σ* en el orden canónico, y a partir de cada w simula M_1. Cuando M_1 acepta w, entonces M_2 lo imprime seguido de «#» en la cinta de output. Ahora las simulaciones se pueden efectuar completamente, dado que M_1 se detiene siempre.

Se probará ahora que si ∃M: L = G(M) y M imprime L sin repetir strings y en el orden canónico, entonces L ε R:

Sea la MT M_1, tal que $G(M_1) = L$ y M_1 imprime los strings de L sin repeticiones y en el orden canónico. Se va a construir una MT M_2, tal que $L(M_2) = L$ y M_2 se detiene siempre.
Como se hizo en la prueba anterior, M_2 tiene una cinta más que M_1, en la que está su input v. M_2 trabaja como M_1, pero toda vez que M_1 imprime un string w, M_2 hace:

- Si w = v, acepta.
- Si w > v según el orden canónico, rechaza (ya no hay posibilidades de que M_1 imprima v).
- Si w < v según el orden canónico, espera por la próxima impresión de M_1 (puede ser que M_1 imprima en el futuro el string v).

Además, si M_1 en algún momento se detiene, entonces M_2 rechaza.

Notar que la prueba es correcta sólo para los casos de lenguajes infinitos. Si L es finito y la MT M_1 que lo genera no se detiene, puede suceder que M_2 no se detenga. Es el caso en que el input v de la MT M_2 es mayor

que el mayor string w de L, según el orden canónico: M_2 esperará indefinidamente que M_1 imprima un string.

Pero si L es finito, entonces se prueba fácilmente que es recursivo, construyendo una MT M_3 que lo reconoce y se detiene siempre.

Por lo tanto, la prueba definitiva de esta segunda parte del teorema queda planteada de la siguiente manera:

- Si L es finito, entonces se prueba que L ε R construyendo la MT M_3.
- Si L es infinito, entonces se prueba que L ε R construyendo la MT M_2.

□

Además de las MT, existe otro modelo en la teoría de los lenguajes formales con la visión de generador de lenguajes, que son las *gramáticas*. El estudio detallado de las gramáticas está fuera del alcance de este libro. En lo que resta de este capítulo y en el capítulo siguiente, se presentan características generales de las gramáticas, por su estrecha relación con las MT.

Definición. Una gramática G es una 4-tupla G = $<V_N, V_T, P, S>$, tal que:

- V_N es el conjunto de los símbolos no terminales de G. Es un conjunto finito.
- V_T es el conjunto de los símbolos terminales de G. Es un conjunto finito. Vale $(V_N \cap V_T) = \varnothing$. Por convención, la unión $(V_N \cup V_T)$ se denota con V.
- P es el conjunto de producciones de G. Una producción es una expresión de la forma $\alpha \rightarrow \beta$, tal que $\alpha \varepsilon V^+$ y $\beta \varepsilon V^*$.
- Finalmente, S es el símbolo inicial de G, donde S ε V_N.

□

La generación de un lenguaje L, a partir de una gramática G, se establece en base a una relación denominada «$\Rightarrow_G$». Si, por ejemplo, $\alpha \rightarrow \beta$ es una producción de G, y los strings γ y δ están en V*, entonces se cumple que: $\gamma\alpha\delta \Rightarrow_G \gamma\beta\delta$.

La expresión $\alpha_1 \Rightarrow^*_G \alpha_m$ denota que $\alpha_1 \Rightarrow_G \alpha_2$, $\alpha_2 \Rightarrow_G \alpha_3$, …, $\alpha_{m-1} \Rightarrow_G \alpha_m$. Se dice en este caso que existe una derivación de α_1 a α_m, o que α_m deriva de α_1.

Definición. El lenguaje L generado por la gramática G, denotado con L(G), es:

$$L(G) = \{w \mid w \in V_T^* \text{ y } S \Rightarrow_G^* w\}.$$

Es decir, L(G) es el conjunto de todos los strings de V_T^* que derivan del símbolo inicial S.

□

Ejemplo. Se cumple que la gramática G = <{S}, {0, 1}, {1. $S \rightarrow 0S1$, 2. $S \rightarrow 01$}, S} genera el lenguaje $L = \{0^n1^n \mid n \geq 1\}$. Los strings de L(G) se obtienen de aplicar cero o más veces la producción 1 y luego una vez la producción 2.

Por la forma de las producciones, las gramáticas se clasifican en cuatro tipos, de acuerdo a una jerarquía conocida como Jerarquía de Chomsky:

- *Gramática de tipo 0*. No tiene ninguna restricción, es la que se definió recién.
- *Gramática de tipo 1 o sensible al contexto*. Toda producción $\alpha \rightarrow \beta$ cumple que $|\alpha| \leq |\beta|$. Una caracterización equivalente, que explica el nombre del tipo de gramática, es que toda producción tiene la forma $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, donde $A \in V_N$, α_1 y $\alpha_2 \in V^*$, y $\beta \in V^+$.
- *Gramática de tipo 2 o libre de contexto*. Toda producción es de la forma $A \rightarrow \alpha$, donde $A \in V_N$, y $\alpha \in V^+$, lo que explica el nombre del tipo de gramática.
- *Gramática de tipo 3 o regular*. Toda producción es de la forma $A \rightarrow aB$ o bien $A \rightarrow a$, donde A y B $\in V_N$, y *a* $\in V_T$.

Los lenguajes se clasifican de la misma manera que las gramáticas que los generan: lenguajes de tipo 0, de tipo 1 o sensibles al contexto, de tipo 2 o libres de contexto, y de tipo 3 o regulares. Se cumple por definición que una gramática (respectivamente un lenguaje) de tipo i, es un caso particular de gramática (respectivamente lenguaje) de tipo k, con i > k.
Para que el string vacío λ pueda ser parte de un lenguaje de tipo 1, se permite el uso de la producción especial $S \rightarrow \lambda$, siempre que S no aparezca en la parte derecha de ninguna producción.
Se prueba que un lenguaje es de tipo 0 sii es recursivamente numerable, y que los lenguajes de tipo 1 constituyen un subconjunto propio de los lenguajes recursivos.

En el siguiente capítulo, se describen MT específicas para reconocer los distintos tipos de lenguajes.

A1.3. Máquinas de Turing restringidas

En la teoría de lenguajes formales se estudian cuatro tipos de MT: las MT generales que se utilizaron a lo largo de toda la primera parte del libro, y tres tipos de MT restringidas, es decir con menor poder computacional en comparación con las MT generales.
Estas MT restringidas reconocen específicamente los lenguajes de tipo 3, 2 ó 1 definidos en el capítulo anterior. Un estudio detallado de las mismas está fuera del alcance de este libro. La idea, en lo que sigue, es presentar sus características generales (Secciones A1.3.1 a A1.3.3). Se destaca en particular, que algunos problemas sobre lenguajes que son indecidibles en el caso general, resultan decidibles cuando se consideran lenguajes de tipo 3, 2 ó 1.

A1.3.1. Autómatas finitos

Definición. Un *autómata finito* (o AF) M es una MT que tiene las siguientes restricciones:

- M tiene una sola cinta, la cinta de input, que es de solo lectura, y sobre la cual el cabezal se mueve solamente a la derecha desde el primer símbolo del input. Cuando el cabezal lee un «B», M se detiene. De este modo, los AF no tienen memoria, salvo la que pueden proveerle los estados.
- Q, Σ, q_0 y F son los mismos componentes que tiene una MT general (del modelo inicial).
- No existe un alfabeto Γ, dado que M sólo tiene una cinta de solo lectura.
- La función de transición δ se define así: δ: Q x $\Sigma \rightarrow$ Q.

□

Ejemplo. Sea el AF M = <Q, Σ, δ, q_0, F>, tal que:

- Q = $\{q_0, q_1, q_2, q_3\}$.
- $\Sigma = \{0, 1\}$.

- $q_0 = q_0$.
- $F = \{q_0\}$.
- La función de transición δ se define de la siguiente manera:

1. $\delta(q_0, 0) = q_2$	2. $\delta(q_0, 1) = q_1$
3. $\delta(q_1, 0) = q_3$	4. $\delta(q_1, 1) = q_0$
5. $\delta(q_2, 0) = q_0$	6. $\delta(q_2, 1) = q_3$
7. $\delta(q_3, 0) = q_1$	8. $\delta(q_3, 1) = q_2$

La función de transición δ se puede describir de una manera alternativa mediante un *diagrama de transición de estados*, que se muestra en la figura A1.3.1.

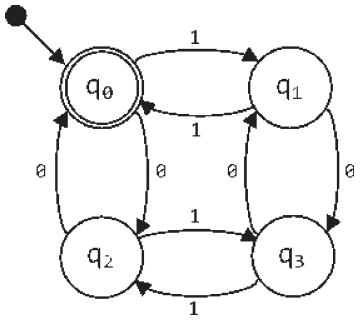


Figura A1.3.1

Los estados se representan por nodos. Existe un arco orientado con nombre *a* del nodo *q* al nodo *p*, si se cumple que $\delta(q, a) = p$. El nodo que representa el estado inicial se señala por una flecha, y los estados finales se representan con nodos con contorno doble.
Se cumple que L(M) es el conjunto de todos los strings de {0, 1}* que tienen un número par de unos y un número par de ceros.

Ejercicio. *Construir una MT general que reconozca el lenguaje del ejemplo.*

□

Se enumeran a continuación algunas características para destacar de los AF:
El modelo de los AF determinísticos es equivalente al modelo de los AF no determinísticos.
Existe otro modelo de AF en los que el cabezal puede moverse a derecha, izquierda o no moverse, sin salirse de los extremos del input. Se conocen como AF *two-way.* Se prueba que no agregan poder computacional.
Dado un AF M, existe una gramática regular G tal que L(M) = L(G). Y dada una gramática regular G, existe un AF M tal que L(G) = L(M). En otras palabras, el poder computacional de los AF alcanza para reconocer todos los lenguajes regulares y sólo ellos.
La clase de lenguajes reconocidos por los AF, es decir los lenguajes regulares, es cerrada con respecto a las operaciones de unión, intersección, complemento, producto, clausura y reversa, e incluye a los lenguajes finitos. Se cumple que la clase de los lenguajes regulares es la más chica que contiene a todos los conjuntos finitos y es cerrada con respecto a la unión, el producto y la clausura.
Dados los AF M y M', los problemas: (a) ¿L(M) = ∅?, (b) ¿L(M) = Σ*?, (c) ¿L(M) es finito?, (d) ¿L(M) es infinito?, (e) ¿L(M) = L(M')?, son decidibles. Se vio antes que todos estos problemas son indecidibles en el caso de las MT generales.

A1.3.2. Autómatas con pila

Definición. Un *autómata con pila* (o AP) M es una MT no determinística que tiene las siguientes restricciones:

- M tiene una cinta de input de solo lectura y una cinta de trabajo que se comporta como una pila (estructura de datos LIFO, por *last-in-first-out*). De este modo, los AP tienen memoria, pero limitada en comparación con las MT generales.
- Hay dos tipos de pasos en un AP: (a) Se lee el estado corriente, el símbolo corriente del input y el tope de la pila, y como consecuencia el estado puede cambiar, el cabezal del input se mueve un lugar a la derecha, y en la pila se agrega un símbolo, se extrae un símbolo, o se reemplaza el símbolo tope. (b) Se lee el estado corriente y el tope de la pila, y como consecuencia el estado puede cambiar, y en la pila se agrega un símbolo, se extrae un símbolo, o se reemplaza

el símbolo tope (por no considerarse el input este movimiento se denomina *λ-movimiento*).
- El AP acepta cuando se vacía la pila. Una definición alternativa de aceptación que se prueba equivalente es la utilizada en los AF, es decir la detención en un estado final.
- Q, Σ, Γ y q_0 son los mismos componentes que tiene una MT general.
- Formalmente, un AP M se define como una 7-tupla M = <Q, Σ, Γ, Δ, q_0, Z_0, F>. Z_0 ε Γ es el símbolo inicial de la pila. Si F = ∅, se asume que hay aceptación por pila vacía, y si no, por estado final. Δ se define así: Δ: Q x (Σ ∪ {λ}) x Γ → $\mathcal{P}$(Q x Γ*).

□

Ejemplo. Sea el AP M = <{q_1, q_2}, {0, 1}, {R, C, U}, Δ, q_1, R, ∅>, siendo Δ:

1. Δ(q_1, 0, R) = {(q_1, CR)}
2. Δ(q_1, 1, R) = {(q_1, UR)}
3. Δ(q_1, 0, U) = {(q_1, CU)}
4. Δ(q_1, 1, C) = {(q_1, UC)}
5. Δ(q_1, 0, C) = {(q_1, CC), (q_2, λ)}
6. Δ(q_1, 1, U) = {(q_1, UU), (q_2, λ)}
7. Δ(q_2, 0, C) = {(q_2, λ)}
8. Δ(q_2, 1, U) = {(q_2, λ)}
9. Δ(q_2, λ, R) = {(q_2, λ)}

Se cumple que L(M) = {ww^R | w ε {0, 1}$^+$}, siendo w^R el string inverso de w.

Se enumeran a continuación algunas características para destacar de los AP:

El modelo de los AP no determinísticos no es equivalente al modelo de los AP determinísticos.
Dado un AP M, existe una gramática libre de contexto G tal que L(M) = L(G). Y dada una gramática libre de contexto G, existe un AP M tal que L(G) = L(M). En otras palabras, el poder computacional de los AP alcanza para reconocer todos los lenguajes libres de contexto y sólo ellos.
La clase de lenguajes reconocidos por los AP, es decir los lenguajes libres de contexto, es cerrada con respecto a las operaciones de unión, producto, clausura y reversa. No es cerrada con respecto a la intersección ni al complemento.

Dados los AP M y M', los problemas: (a) ¿L(M) = ∅?, (b) ¿L(M) es finito?, (c) ¿L(M) es infinito?, son decidibles, y los problemas: (d) ¿L(M) = Σ*?, (e) ¿L(M) = L(M')?, son indecidibles. Para el caso de los AP determinísticos, el problema (d) es decidible, mientras que no se sabe si el problema (e) es decidible o no.

A1.3.3. Autómatas acotados linealmente

Definición. Un *autómata acotado linealmente* (o AAL) M es una MT no determinística con una sola cinta, sobre la que el cabezal nunca se mueve más allá de los extremos del input. Formalmente, los AAL se definen como las MT generales (modelo inicial):

$$M = <Q, \Sigma, \Gamma, \Delta, q_0, F>.$$

La relación de transición es Δ: Q x $\Gamma \rightarrow \mathcal{P}$(Q x Γ x {L, R, S}). Σ tiene dos símbolos especiales, «#» y «$», que se utilizan como las marcas izquierda y derecha del input, respectivamente.

□

Ejemplo. La MT descripta al comienzo del primer capítulo del libro, para reconocer el lenguaje $L = \{a^n b^n \mid n \geq 1\}$, es un ejemplo de AAL, en particular determinístico.

Dado un AAL M, existe una gramática sensible al contexto G tal que L(M) = L(G). Y dada una gramática sensible al contexto G, existe un AAL M tal que L(G) = L(M). En otras palabras, el poder computacional de los AAL alcanza para reconocer todos los lenguajes sensibles al contexto y sólo ellos.
Se prueba que los lenguajes sensibles al contexto son recursivos (por lo tanto también lo son los lenguajes libres de contexto y los lenguajes regulares). La demostración se basa en que la parte derecha de toda producción de una gramática de tipo 1, no mide menos que la parte izquierda, por lo que chequear si existe una derivación desde el símbolo inicial a un string de símbolos terminales es decidible.
También se prueba que los lenguajes sensibles al contexto son un subconjunto propio de R:

Teorema. Existen lenguajes recursivos que no son sensibles al contexto.

Prueba. La prueba se basa en la técnica de diagonalización. Al igual que las MT, las gramáticas también pueden codificarse y ordenarse (en particular las de tipo 1). Queda como ejercicio para el lector, proponer una codificación para las gramáticas de tipo 1. Sea el lenguaje $L = \{w_i \mid w_i \notin L(G_i)\}$, donde G_i es la gramática de tipo 1 i-ésima, según el orden canónico.
Se cumple que L es recursivo. Dado w_i, se determina i según el orden canónico, se genera G_i, se chequea si w_i está o no en $L(G_i)$, y se rechaza o acepta, respectivamente.
Se cumple que L no es sensible al contexto. Supóngase que lo es, es decir que L se genera por una gramática G_k de tipo 1. Si $w_k \in L$, entonces $w_k \in L(G_k)$, pero por la definición de L se cumple que $w_k \notin L(G_k)$. Si $w_k \notin L$, entonces $w_k \notin L(G_k)$, pero por la definición de L se cumple que $w_k \in L(G_k)$. Por lo tanto, L es recursivo y no es sensible al contexto.

□

Otras características para destacar de los AAL son las siguientes:
No se sabe si el modelo de los AAL no determinísticos es equivalente al modelo de los AAL determinísticos.
La clase de lenguajes reconocidos por los AAL, es decir los lenguajes sensibles al contexto, es cerrada con respecto a las operaciones de unión, intersección, producto, clausura y reversa. No se sabe si es cerrada con respecto al complemento.
Dados los AAL M y M', los problemas: (a) ¿$L(M) = \varnothing$?, (b) ¿L(M) es finito?, (c) ¿L(M) es infinito?, (d) ¿$L(M) = \Sigma^*$?, (e) ¿$L(M) = L(M')$?, son indecidibles, tal como sucede con las MT generales.

Referencias bibliográficas de la Parte I

En 1936, A. Turing definió las *automatic machines* (máquinas de Turing). Argumentando que dicho modelo podía llevar a cabo cualquier algoritmo, demostró la existencia de problemas indecidibles (Turing, 1936). Se definieron otros modelos computacionales, como por ejemplo las funciones recursivas parciales (Kleene, 1936), el Lambda cálculo (Church, 1936), y los sistemas de Post (Post, 1936). Todos resultaron ser equivalentes entre sí.

La caracterización de los conjuntos R y RE que se enuncia en el Teorema de Rice se debe a H. Rice (Rice, 1953). La demostración del teorema puede encontrarse en (Hopcroft y Ullman, 1979). Existen otras formas de relacionar a R con RE, como la que se presenta en (Håstad, 1999). Con la misma técnica de prueba utilizada en (Turing, 1936), se llega a la conclusión de que todo sistema que puede expresar universalidad posee lenguajes indecidibles (Morales Luna, 2000).

En (Gács y Lobas, 1999) y (Arora y Barak, 2007) se presentan varios modelos alternativos de maquinas de Turing equivalentes al modelo estándar. En la publicación de Arora y Barak, en particular, se incluye la simulación de una máquina con K cintas por una máquina con una cinta sin la utilización de tracks, se presentan definiciones alternativas de las máquinas de Turing no determinísticas, y se introduce la noción de máquinas de Turing olvidadizas. En estas máquinas, el movimiento del cabezal sólo depende de la longitud del input. El trabajo muestra la existencia de máquinas de Turing universales con dicha propiedad. Las máquinas RAM son estudiadas formalmente en (Cook y Reckhow, 1973). Un tratamiento detallado de las máquinas RAM se puede encontrar en (Lewis y Papadimitriou, 1998). En (Chandra, Kozen y Stockmeyer, 1981) se introducen las máquinas de Turing alternantes, las cuales se utilizan como modelo canónico en (Tompa, 1991). Los modelos determinísticos y no determinísticos de MT se pueden definir como casos particulares de dichas máquinas. Las máquinas de Turing alternantes se tratan suscintamente en el apéndice de la segunda parte de este libro, en el marco de la complejidad computacional. Otra formulación interesante de máquinas de Turing puede encontrarse en (Wolfram, 2002).

Los autómatas finitos son utilizados ampliamente, por ejemplo para el procesamiento de texto y la implementación de compiladores. Para un estudio detallado de lenguajes, gramáticas y autómatas recomendamos por ejemplo (Hopcroft y Ullman, 1969) y (Sipser 2006). Ambos libros abarcan diferentes modelos de autómatas, y su relación con las gramáticas y expresiones regulares. La jerarquía de clases de lenguajes

conocida como Jerarquía de Chomsky fue definida en (Chomsky, 1956). La prueba de que los lenguajes generados por las gramáticas de tipo 0 coinciden con el conjunto de los lenguajes recursivamente enumerables aparece en (Chomsky, 1959).

Hemos utilizado el lenguaje L_U como punto de partida para demostrar que otros lenguajes son indecidibles. El problema de la correspondencia de Post (Post, 1946) es otro problema indecidible que resulta de gran utilidad para demostrar la indecidibilidad en el área de los lenguajes formales. Existe un teorema análogo al Teorema de Rice en el marco de los lenguajes libres de contexto, denominado Teorema de Greibach (Greibach, 1968). Con este teorema es posible probar, por ejemplo, que es indecidible determinar si el lenguaje generado por una gramática libre de contexto es regular.

Recomendamos la lectura de los libros (Penrose, 1989) y (Herken, 1994). En el primero, el matemático R. Penrose cuestiona los postulados de la inteligencia artificial, y para ello profundiza no sólo en la física y la biología, sino también en la teoría de la computación. El libro de Herken reunió una importante cantidad de artículos de los autores más destacados de la teoría de la computación, cuando se cumplieron 50 años desde la aparición del artículo fundacional de A. Turing.

A continuación se lista la que consideramos bibliografía principal en relación a la primera parte del libro. En este sentido, (Hopcroft y Ullman, 1979) y (Hopcroft y Ullman, 1969) tratan en conjunto varios de los temas que hemos desarrollado:

- Arbib M. [1964]. Brains, Machines and Mathematics. New York: Mc Graw-Hill Book Company.
- Arora S. y Barak B. [2007]. Computational Complexity: A Modern Approach. Princeton University.
- Brady J. M. [1977]. The Theory of Computer Science. A programming approach. Science paperbacks.
- Chaitín G. [1974]. Information - Theoretic Limitations of Formal Systems. ACM 21, 403.
- Chandra A. K.. Kozen D. C. y Stockmeyer L. J [1981]. Alternation. Journal of ACM, 28, 114-133.
- Chomsky N. [1956]. Three models for the description of language. IRE Transactions on Information Theory 2, 3. pp. 113-124.
- Chomsky N. [1959]. On certain formal properties of grammars. Information and Control 2, 137-167.
- Church A. [1936]. An unsolvable problem of elementary number theory. Amer. J. Math. 58, 345-363.
- Cook S. A. y Reckhow R. A. [1973]. Time bounded random access machines. J. Computer and System Sciences 7:4, 354-375.

- Davis M., ed. [1958]. Computability and Unsolvability New York: McGraw-Hill.
- Gács P. y Lovász L. [1999]. Complexity of Algorithms. Lecture Notes.
- Greibach S. A. [1968]. A note on undecidable properties of formal languages. Math Systems Theory 2:1, pp. 1-6.
- Hamilton A. [1978]. Logic for Mathematicians. Cambridge University Press.
- Håstad J. [1999]. Complexity Theory. Royal Institute of Thechnology.
- Herken R. (Ed.) [1994]. The Universal Turing Machine. A Half-Century Survey. Springer-Verlag.
- Hermes H. [1969]. Enumerability, decidability, computability. Springer-Verlag.
- Hopcroft J. E. y Ullman J. D. [1969]. Formal Languages and their Relation to Automata. Addison-Wesley, Reading, Mass.
- Hopcroft J. E. y Ullman J. D. [1979]. Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, Mass.
- Kleene S. C. [1936]. General recursive functions of natural numbers. Mathematische Annalen 112, 727-742.
- Kleene S. C. [1962]. Introduction to Metamathematics. North-Holland.
- Kozen D. C. [1997]. Automata and Computability. Springer-Verlag, Berlin.
- Lewis H. R. y Papadimitriou C. H. [1998]. Elements of the Theory of Computation. Prentice-hall.
- Markov A. A. [1954]. Theory of algorithms. Trudy Mathematicheskogo Instituta imeni V. A. Steklova, 42.
- Mendelson E. [1964]. Introduction to Mathematical Logic. Van Nostrand.
- Minsky M. L. [1967]. Computation: Finite and Infinite Machines, Englewood Cliffs, N.J., Prentice-Hall.
- Morales Luna G. [2000]. Computabilidad y Complejidad, CINVESTAV-IPN, México.
- Moret B. M. E. [1998]. The Theory of Computation. Addison-Wesley-Longman.
- Penrose R. [1989]. The Emperor's New Mind. Oxford University Press.
- Post E. [1936]. Finite combinatory processes-formulation. J. Symbolic Logic 1, 103-105.
- Post E. [1946]. A variant of a recursively unsolvable problem. Bull. AMS, 52, 264-268.
- Rice H. G. [1953]. Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc. 74, 358-366.
- Rogers H. [1967]. Theory of Recursive Functions and Effective Computability. McGraw-Hill.
- Sagastume M. y Baum G. [1986]. Problemas, Lenguajes y Algoritmos. Editora da Unicamp, EBAI.
- Salomaa A. [1973]. Formal languages. Academia Press, New York.
- Shepherdson, J. C. y Sturgis H. E. [1963]. Computability of recursive functions. Journal of ACM, 10, 217-255.
- Sipser M. [2006]. Introduction to the Theory of Computation, Thomson Course Technology.
- Tompa M. [1991]. Introduction to Computational Complexity. University of Whashington.

- Trakhtenbrot B. A. [1973]. Algoritmos y computadoras. Limusa.
- Turing A. M. [1936]. On computable numbers, with an application to the Entscheidungsproblem. Proceedings, London Mathematical Society, 2, 42 pp. 230-265, and no. 43, pp. 544-546.
- van Heijenoort J. [1967]. From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931. Cambridge, Mass.
- Veloso P. [1982]. Aspectos de una teoría general de problemas. Dpto. Informática, PUC-RJ, Tech. Rept.
- Winklmann K. y Schauble C. J. C. [2002]. Theory of Computation Class Notes, September.
- Wolfram S. [2002]. A New Kind of Science. Wolfram Media, Inc.

Ejercicios de la Parte I

E.I.1 Construir máquinas de Turing (MT) que reconozcan los siguientes lenguajes (en todos los casos, x e y son números naturales representados en binario):

a) $L = \{x\#y_1\#...\#y_k \mid x = y_i \text{ para algún } i\}$.

b) $L = \{x\#y \mid x = 2^y\}$.

c) $L = \{x\#y \mid x! = y\}$.

E.I.2 Sea $L = \{a^n b^n c^n \mid n \geq 0\}$. Construir la MT más eficiente posible, con respecto a la cantidad de pasos ejecutados, que reconozca L.

E.I.3 Construir una MT con una cinta de output que escriba todos los strings de {0,1}* en orden canónico.

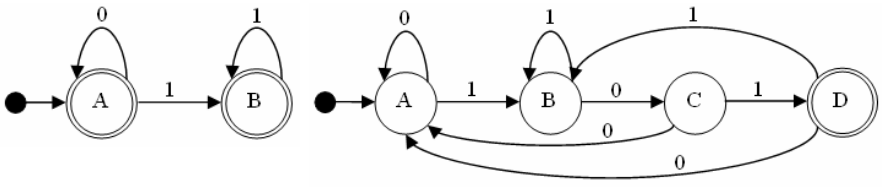
E.I.4 Sea USAT = {ϑ | ϑ es una fórmula booleana que tiene exactamente una asignación que la satisface}. ¿Se cumple que la siguiente MT no determinística acepta USAT?

(a) Rechazar si ϑ no es un input válido.
(b) Seleccionar no determinísticamente una asignación A.
(c) Rechazar si A no satisface ϑ.
(d) Seleccionar no determinísticamente una asignación A' $\neq$ A.
(e) Rechazar sii A' satisface ϑ.

E.I.5 Construir autómatas finitos que acepten los siguientes lenguajes de {0,1}*:

a) Todos los strings con tres ceros consecutivos.
b) Todos los strings tales que por cada cero haya un uno inmediatamente a la derecha.

E.I.6 Describir los lenguajes aceptados por los autómatas finitos cuyos diagramas de transición de estados se muestran a continuación:



E.I.7 Construir un autómata con pila que acepte el lenguaje de {0,1}* de todos los strings que tengan igual cantidad de ceros y unos.

E.I.8 Determinar las diferencias existentes entre las MT restringidas descriptas en el libro y las MT generales, y argumentar en cada caso la razón por la que las MT generales poseen mayor poder computacional.

E.I.9 Hacer el mismo análisis comparativo que en el ejercicio anterior, pero ahora considerando las MT restringidas entre sí.

E.I.10 Probar que para toda MT $M_1 = <Q_1, \Sigma_1, \Gamma_1, \delta_1, q_1, q_A, q_R>$, con

$$\delta_1: Q_1 \times \Gamma_1 \rightarrow (Q_1 \cup \{q_A, q_R\}) \times \Gamma_1 \times \{L, R, S\},$$

existe una MT equivalente $M_2 = <Q_2, \Sigma_2, \Gamma_2, \delta_2, q_2, q_A, q_R>$, con

$$\delta_2: Q_2 \times \Gamma_2 \rightarrow (Q_2 \cup \{q_A, q_R\}) \times \Gamma_2 \times \{L, R\}.$$

E.I.11 Las MT con «salto a la izquierda» tienen una función de transición de la forma:

$$\delta: Q \times \Gamma \rightarrow (Q \cup \{q_A, q_R\}) \times \Gamma \times \{R, JUMP\}, \text{ tal que:}$$

si $\delta(q, a) = (q', a', JUMP)$, entonces la máquina pasa al estado q', escribe el símbolo a', y posiciona el cabezal sobre el símbolo de más a la izquierda, quedándose en el mismo lugar si a la izquierda hay sólo símbolos blancos. Probar que dicho modelo es equivalente al estándar.

E.I.12 Probar que es posible simular una MT M del modelo estándar con otra que escribe en una celda a lo sumo una vez.

E.I.13 Supóngase que se cambia la condición de aceptación y rechazo de un input de la siguiente manera: un input es aceptado por una MT M sii M finaliza su ejecución y todas sus cintas tienen únicamente símbolos blancos. Probar que para toda MT del modelo estándar existe una MT equivalente del nuevo modelo.

E.I.14 Se define un modelo alternativo de MT según el cual una máquina puede, además de alterar el contenido de las celdas, eliminar un símbolo existente removiendo la celda de la cinta, e insertar un nuevo símbolo agregando una nueva celda a la cinta. Probar que dicha máquina puede ser simulada por una MT estándar.

E.I.15 Se define un modelo alternativo de MT, tal que existen k cabezales por cinta, siendo $k > 1$. Probar que con el modelo estándar se puede simular el nuevo modelo.

E.I.16 Determinar si para toda MT $M_1 = <Q_1, \Sigma_1, \Gamma_1, \delta_1, q_1, q_A, q_R>$, existe una MT equivalente $M_2 = <Q_2, \Sigma_2, \Gamma_2, \delta_2, q_2, q_A, q_R>$ tal que al comienzo no se sabe a qué celda apunta su cabezal al input.

E.I.17 Determinar la verdad o falsedad de los siguientes enunciados:

a) Hay igual cantidad de MT que de lenguajes en el conjunto CO-RE.
b) Hay igual cantidad de MT determinísticas que de MT no determinísticas.
c) Si L es un lenguaje recursivamente numerable, y $L' \subseteq L$, entonces L' es un lenguaje recursivamente numerable.

E.I.18 Sean L_1 y $L_2 \in$ RE. Determinar si $(L_1 - L_2) \in$ RE.

E.I.19 Sean L_1 y $L_2 \in$ CO-RE. Determinar si $(L_1 \cap L_2) \in$ CO-RE.

E.I.20 Probar que si L ε RE, entonces existe una MT que lo lista sin repetir elementos.

E.I.21 Construir una MT que imprima en su cinta de output los códigos de todas las MT que, a partir del input vacío, se detienen en a lo sumo 100 pasos.

E.I.22 Sea $L \in$ RE. Probar que para todo $k \geq 0$ vale $L^k \in$ RE, siendo $L^0 = \{\lambda\}$, y $L^i = L^{i-1} \bullet L$ para todo $i \geq 1$.

E.I.23 Sean S y T lenguajes recursivamente numerables de $\{0,1\}^+$. Probar que también es recursivamente numerable el lenguaje $U = \{x \varepsilon \{0,1\}^+ \mid \exists y \in S, z \in T, x = y + z\}$.

E.I.24 Probar que si $L_1 \alpha L_2$, entonces también $L_1^C \alpha L_2^C$.

E.I.25 Probar que $L_U \alpha$ HP.

E.I.26 Determinar si los siguientes lenguajes pertenecen a las clases R, (RE – R), (CO-RE – R), o $\mathcal{L}$ – (RE $\cup$ CO-RE):

a) {<M> | $\exists$w tal que M se detiene a partir de w en a lo sumo 1000 pasos}.
b) {<M> | $w \in$ L(M) tal que |w| < 100}.
c) {<M> | L(M) es finito}.
d) {<M> | $L(M) \neq \Sigma^*$}.
e) {<M> | L(M) = S}, con $S \subseteq \Sigma^*$.
f) {<M> | $\lambda \in$ L(M)}.
g) {<M> | $w \in$ L(M) sii $w^R \in$ L(M)}.
h) {(<M_1>, <M_2>) | $L(M_1) \neq L(M_2)$}.
i) {(<M_1>, <M_2>) | $L(M_1) \subseteq L(M_2)$}.

j) {<M> | L(M) $\in$ RE}.
k) {<M> | M es una MT con 14 estados}.

E.I.27 Resolver el ejercicio anterior utilizando el Teorema de Rice.

E.I.28 ¿Cuáles de los siguientes lenguajes pertenecen a RE?
a) POCOS = {<M>| |L(M)| $\leq$ 2}.
b) MUCHOS = {<M>| |L(M)| $\geq$ 2}.

E.I.29 Sea f: $\Sigma^* \rightarrow$ {0, 1} tal que:
f(x) = 1, si x = (<M>, w) y M acepta w.
f(x) = 0, si x = (<M>, w) y M no acepta w, o bien si x $\neq$ (<M>, w).
Determinar si f es una función total computable.

E.I.30 Dado w ε (Σ^* - {λ}), se define como E(w) al string generado a partir de w reemplazando cada cero por un uno y cada uno por un cero. Por ejemplo, E(1001) = 0110. Un lenguaje A incluido en Σ^* es un lenguaje espejo si cumple que $\forall$w ε ($\Sigma^* - \{\lambda\}$): w ε A sii E(w) pertenece a A^C. Probar que si A es un lenguaje espejo y A no pertenece a RE, entonces A^C tampoco pertenece a RE.

E.I.31 El problema de la detención de las MT yendo sólo a la derecha es el mismo que el problema de la detención de las MT, salvo que considera el modelo de las MT que se mueven sólo a la derecha. Definir el lenguaje HPD asociado al problema, y determinar si en este caso HPD es recursivo.

E.I.32 El problema de la detención universal es el mismo que el problema de la detención de las MT, salvo que considera la detención de una MT con respecto a todos los strings de Σ^*. Definir el lenguaje HPU asociado al problema, y determinar si en este caso HPU es recursivo.

E.I.33 Sea el lenguaje HPE = {<M> | $\exists$w tal que M se detiene a partir de w}. Probar que HPE $\notin$ R.

E.I.34 Sea el siguiente problema: una MT M tiene una cinta de input y una cinta de output, el alfabeto de sus inputs es $\Sigma = \{X\}$, el alfabeto de sus cintas es $\Gamma = \{X, B\}$, y la pregunta asociada al problema es si acaso M escribe alguna vez «X» cuando su input es λ. ¿Se cumple que este problema es decidible?

E.I.35 Un estado inalcanzable en una MT M, es un estado al que nunca llega M. Considerar el problema de determinar si una MT tiene un estado inalcanzable. ¿Acaso este problema es decidible?

E.I.36 Considerar el problema de determinar si una MT siempre escribe un símbolo blanco sobre un símbolo no blanco. ¿Acaso este problema es decidible?

E.I.37 Un lenguaje infinito L es recursivamente numerable sin repeticiones, si L es el codominio de una función computable inyectiva. Probar que L es recursivamente numerable sin repeticiones, sii L es infinito y recursivamente numerable.

E.I.38 Un lenguaje L es recursivamente numerable en orden creciente, si L es el codominio de una función total computable creciente. Una función f es creciente si, dada la relación <, para todo par x, y, si $x < y$ entonces $f(x) < f(y)$. Probar que L es recursivamente numerable en orden creciente, sii L es infinito y recursivo.

E.I.39 Determinar si los siguientes lenguajes pertenecen o no pertenecen a RE:

a) IMP = $\{(<M_1>, <M_2>) \mid |L(M_1) - L(M_2)| \text{ es impar}\}$.

b) COMP = {<M> | L(M) y el complemento de L(M) son infinitos}.

c) COD = {<M> | L(M) = {<M>}}.

E.I.40 ¿Se cumple que todo lenguaje infinito perteneciente a RE tiene un subconjunto infinito perteneciente a R?

E.I.41 ¿Se cumple que todo lenguaje infinito perteneciente a R tiene un subconjunto no perteneciente a RE?

E.I.42 ¿Se cumple que todo lenguaje infinito perteneciente a RE tiene un subconjunto no perteneciente a RE?

PARTE II
COMPLEJIDAD COMPUTACIONAL

Introducción de la Parte II

Ya dejamos atrás los problemas indecidibles. En este segundo tramo de nuestro viaje imaginario, nos acercaremos más a los problemas decidibles. Ahora estaremos recorriendo problemas intratables y problemas tratables. Alcanzaremos una distancia que nos va a permitir observar cómo dichos problemas se distribuyen en este microuniverso…

El tema de la segunda parte del libro es la *complejidad computacional*. Ya caracterizados los problemas decidibles, ahora se los va a estudiar desde el punto de vista de los recursos que requiere una máquina de Turing para resolverlos, específicamente el tiempo (cantidad de pasos) y el espacio (cantidad de celdas) consumidos. Existen otras medidas posibles de complejidad, como la cantidad de veces que en la ejecución de una máquina de Turing con una sola cinta, el cabezal cambia de dirección. También se puede encarar el estudio de una manera más abstracta, definiendo una axiomática relacionada con algún recurso genérico.
Otras formulaciones se relacionan con la complejidad estructural de las máquinas de Turing, por ejemplo teniendo en cuenta la cantidad de estados, la cantidad de símbolos, o el tipo de cintas utilizadas. Particularmente, la teoría de los lenguajes formales clasifica a los lenguajes regulares como menos complejos que los lenguajes libres de contexto, porque los autómatas finitos, que reconocen a los primeros, son menos complejos estructuralmente que los autómatas con pila, que reconocen a los otros. El mismo tipo de comparación se puede establecer considerando los lenguajes sensibles al contexto y los lenguajes recursivamente numerables.

Históricamente, una de las primeras medidas de complejidad fue la de C. Shannon, en 1956, que consideraba el producto entre la cantidad de estados y la cantidad de símbolos de una máquina de Turing. En 1960, M. Rabin formuló las bases de la teoría de la complejidad abstracta que desarrolló M. Blum en 1967. J. Hartmanis y R. Stearns fueron quienes en 1965 introdujeron la expresión *complejidad computacional*, y la definieron como el tiempo consumido por una máquina de Turing de varias cintas. En el mismo año 1965, A. Cobham identificó la clase de problemas que se podían resolver en tiempo polinomial con respecto a la longitud de los inputs. Esta clase era interesante porque no dependía del modelo de ejecución. También en 1965, F. Hennie publicó su trabajo sobre las secuencias de cruces en las máquinas de Turing con una sola cinta.

Nuestro enfoque apunta a razonar sobre la complejidad intrínseca de los problemas, independientemente de los algoritmos que existan para resolverlos. Nos basaremos en el tiempo y el espacio consumidos por una máquina de Turing, por ser las medidas de complejidad más populares e intuitivas. Será esencial definir un modelo de máquina de Turing y un modelo de representación de datos «consistentes» para el estudio de la complejidad, lo que se va a precisar después. La idea es analizar el tiempo y el espacio en función del tamaño de los datos de entrada, obteniendo su razón de crecimiento (logarítmica, lineal, polinomial, exponencial). De este análisis surgirá que determinados problemas, a pesar de ser decidibles, resultan en la práctica intratables por la gran cantidad de recursos que demandan.

Los Capítulos 2.1 y 2.2 tratan la complejidad computacional temporal. En el Capítulo 2.1 se introducen las definiciones fundamentales, y se presentan las características generales de la jerarquía temporal de problemas.

En el Capítulo 2.2 se estudian las clases temporales de problemas más importantes. Esencialmente, se identifica la clase de los problemas tratables como la de los problemas que se resuelven en tiempo determinístico polinomial, la clase P. También se caracteriza la clase de los problemas que se resuelven en tiempo no determinístico polinomial, la clase NP, por incluir importantes problemas de la ciencia de la computación y las matemáticas, y se describe su estructura. En este marco se introduce el concepto de NP-completitud, mediante el cual, asumiendo la conjetura de que hay más problemas que se resuelven en tiempo no determinístico polinomial que en tiempo determinístico polinomial, es decir $P \subset NP$, se comprueba la intratabilidad de importantes problemas.

Simétricamente, en el Capítulo 2.3 se presentan las definiciones fundamentales de la complejidad espacial y las características generales de la jerarquía espacial de problemas, y en el Capítulo 2.4 se tratan clases espaciales específicas.
En el Capítulo 2.4, se identifica la clase de los problemas tratables como la de los problemas que se resuelven en espacio logarítmico. Se extiende el concepto de completitud a cualquier clase de complejidad. Y se establecen relaciones entre las jerarquías temporal y espacial, llegando a definirse una única jerarquía espacio-temporal.
Finalmente, con foco en la complejidad temporal, en el Apéndice 2 se describen suscintamente modelos de máquinas alternativas a las máquinas de Turing, y se analiza cómo las jerarquías de clases asociadas se relacionan con las que se definieron en los capítulos precedentes. Por otro lado, también se analiza la complejidad computacional de problemas más generales que los de decisión.

CAPÍTULO 2.1. INTRODUCCIÓN A LA COMPLEJIDAD TEMPORAL

Dado que las instancias de un problema pueden ser de cualquier tamaño, y así en general la máquina de Turing que lo resuelve debería tardar más (efectuaría más pasos) tratando con inputs grandes que con inputs pequeños, resulta natural permitir que el tiempo de ejecución crezca con el tamaño de los inputs. La idea es, entonces, medir el tiempo de una máquina de Turing por medio de una función temporal que, dado el tamaño n del input, defina el tiempo en términos de n. Se analiza la razón de crecimiento de la función temporal, categorizando la complejidad temporal del problema asociado según si la máquina de Turing que lo resuelve tarda, por ejemplo, a lo sumo un tiempo lineal, polinomial o exponencial.

Se va a justificar enseguida que en las mediciones de la complejidad temporal es irrelevante considerar variaciones de sumandos y factores constantes. De esta manera, se trabaja con órdenes de magnitud. En lugar de funciones f se utilizan funciones O(f), que se leen «orden de f». La expresión O(f) denota el conjunto de las funciones g tales que para todo número natural $n \geq 0$, se cumple que $g(n) \leq c.f(n)$, siendo c una constante. Todo las consideraciones anteriores aplican también al espacio, y serán tomadas en cuenta en los capítulos correspondientes.

En este primer capítulo de la segunda parte del libro se introducen, al comienzo, las definiciones fundamentales relacionadas con la complejidad computacional temporal (Sección 2.1.1). Se precisa qué es el tiempo de trabajo de una máquina de Turing, y cómo se caracteriza una clase de problemas desde el punto de vista de la complejidad temporal.

En la Sección 2.1.2 se introduce la jerarquía de las clases temporales de problemas. Se describen sus características principales, y se mencionan las clases más importantes. En particular, se traza la frontera entre los problemas tratables y los problemas intratables, desde el punto de vista del tiempo. Anticipándonos: los problemas tratables son los que se pueden resolver con máquinas de Turing que trabajan en tiempo determinístico polinomial. Esta clase se denomina P. La otra clase que se destaca es NP, porque nuclea importantes problemas de la ciencia de la computación y las matemáticas. Los problemas de NP se resuelven por máquinas de Turing que trabajan en tiempo no determinístico polinomial, incluyen a P por definición, y no se sabe si la inclusión es estricta.

También se elige y justifica con qué modelo de máquinas de Turing se va a trabajar, y cómo se van a representar los datos. El análisis de la complejidad temporal se mantiene dentro del marco de los problemas

de decisión. En el Apéndice 2 se trata la complejidad temporal de problemas más generales.

2.1.1. Definiciones iniciales

Ya se hizo referencia en la primera parte del libro, al tiempo de trabajo de una MT. El tiempo consumido por una MT M se mide en términos de la cantidad de pasos que M ejecuta. Formalmente:

Definición. Sea la función T: $N \rightarrow N^+$. Una MT M trabaja en tiempo T(n) sii para todo input w, tal que |w| = n, M hace a lo sumo T(n) pasos (en su única computación si M es una MT determinística, o en cada una de sus computaciones si M es una MT no determinística).

□

De modo similar, se puede definir una MT que trabaja en tiempo O(T(n)). El codominio de la función T es N^+ porque toda MT hace al menos un paso. Desde el punto de vista de la complejidad temporal, los lenguajes que pueden ser reconocidos por MT que trabajan en un determinado tiempo O(T(n)), se van a agrupar en una misma clase o conjunto de lenguajes:

Definición. Un lenguaje (o problema) pertenece a la clase DTIME(T(n)), sii existe una MTD M con K cintas que lo reconoce (o resuelve) en tiempo O(T(n)). La misma definición vale para la clase NTIME(T(n)) con respecto a las MTN.

□

En palabras, DTIME(T(n)) es la clase de los problemas que se pueden resolver con MT determinísticas con K cintas que trabajan a lo sumo en tiempo cT(n), para cualquier constante c > 0, siendo n la longitud de los inputs. Lo mismo vale para NTIME(T(n)), pero en términos de MT no determinísticas con K cintas.
Notar que estas definiciones se refieren a cotas superiores de tiempo. En términos de las mismas se desarrollarán los distintos aspectos de la complejidad temporal. También se pueden analizar cotas inferiores de tiempo, y tiempos promedio considerando la distribución probabilística de la población de los inputs, lo que en general es bastante difícil.
También notar que DTIME(T(n)) y NTIME(T(n)) nuclean problemas que se resuelven en tiempo a lo sumo $c_1T(n)$, ó $c_2T(n)$, ó $c_3T(n)$, etc. La no distinción de las constantes quedará justificada más adelante.

Las MT estándar para el análisis de la complejidad temporal serán las MTD con K cintas. Se mostró en la primera parte del libro, que pasar de una MT con K cintas a una MT equivalente con una cinta puede aumentar el tiempo de trabajo en el orden cuadrático, es decir $O(n^2)$. Se puede probar en particular, que reduciendo de K a 2 cintas, el retardo disminuye a $O(n \log_2 n)$. Como se verá después, este posible retardo no invalida la adopción del modelo de las MTD con K cintas como MT estándar.

Ejemplo. Sea el lenguaje que se presentó en la primera parte del libro: $L = \{w \mid w \varepsilon \{a, b\}^*$ y w es un palíndromo$\}$.

Una posible MTD con una cinta para reconocer L es la que a partir de $w = w_1 w_2 \ldots w_k$ hace:

1) Si $w = \lambda$, M acepta.
2) Lee el primer símbolo, x, y si no es «a» ni «b» rechaza. En caso contrario elimina x, y va a la derecha hasta el último símbolo, x'. Si x' no existe, entonces M acepta. Si x' existe y $x \neq x'$, entonces M rechaza.
3) Elimina el último símbolo, vuelve a la izquierda hasta el nuevo primer símbolo, y comienza el ciclo otra vez a partir de (1).

Siendo $|w| = n$, M hace unos $[n + (n - 1) + (n - 2) + \ldots + 2 + 1] = n(n+1)/2$ pasos. Por lo tanto, M trabaja en tiempo $T(n) = O(n^2)$, es decir que $L \varepsilon DTIME(n^2)$.

La siguiente MTD M' con dos cintas reconoce L en menos tiempo que M (esta solución ya fue descripta antes). Dado $w = w_1 w_2 \ldots w_k$, la MT M' hace:

1) Copia w de la cinta 1 a la cinta 2. Si detecta un símbolo distinto de «a» y «b» rechaza. Si no, queda apuntando al primer símbolo de w de la cinta 1 y al último símbolo de w de la cinta 2.
2) Compara los símbolos apuntados. Si son distintos, M' rechaza. Si son iguales y «B», M' acepta. En otro caso, se mueve una celda a la derecha en la cinta 1 y una celda a la izquierda en la cinta 2, y repite (2).

M hace unos 3n pasos, primero n para la copia, luego n para apuntar al primer símbolo del string en la primera cinta, y finalmente otros n para comparar los contenidos de las dos cintas. Por lo tanto, la MT M' trabaja en tiempo $T(n) = O(n)$, es decir que $L \varepsilon DTIME(n)$.

2.1.2. Generalidades de la jerarquía temporal

El propósito de esta sección es presentar las características generales de una jerarquía de clases de problemas, desde el punto de vista de la complejidad temporal. En el siguiente capítulo se estudiarán con bastante detalle determinadas clases de la jerarquía.
Dentro del conjunto R de los problemas decidibles, se distinguen dos clases temporales: P y NP. La clase P agrupa 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. Formalmente:

Definición. $P = \bigcup_{i \geq 0} DTIME(n^i)$, y $NP = \bigcup_{i \geq 0} NTIME(n^i)$.

□

Se cumple por definición que $P \subseteq NP$. Además se prueba que las clases P y NP están incluidas estrictamente en R. En cambio no se sabe si $P \subset NP$, lo que constituye uno de los más importantes problemas abiertos de la teoría de la complejidad computacional. NP se caracteriza por incluir importantes problemas relacionados con la ciencia de la computación y las matemáticas.
Asumiendo que $P \neq NP$, en la figura 2.1.1 se muestra una primera versión de lo que en este libro se identificará como *mapa de la complejidad temporal*:

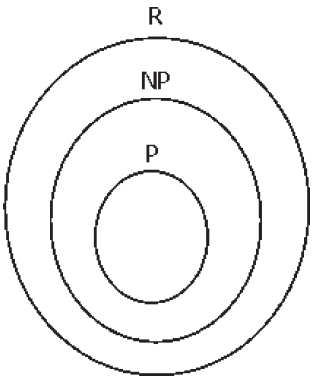


Figura 2.1.1

Se considera a P la clase de los problemas *tratables*, en el sentido de que si bien todos los problemas de R son decidibles, las soluciones que no son determinísticas polinomiales no se consideran eficientes en términos de tiempo.
Esta convención no es arbitraria, se corresponde con la realidad. Podría argumentarse alguna inconsistencia en el caso de inputs pequeños, en que n^k puede superar ampliamente a d^{hn} dadas las constantes k, d y h (supóngase, por ejemplo, n^{1000} vs $2^{0,0001n}$). Pero estos órdenes no suceden en la práctica. Además, la complejidad computacional trata con inputs de todos los tamaños, y las funciones polinomiales, cuando n tiende a infinito, se mantienen de un modo pronunciado por debajo de las funciones exponenciales.
Por oposición, (R – P) será la clase de los problemas *intratables*. Así, la frontera de P define la barrera entre los problemas con solución eficiente e ineficiente. Para simplificar la presentación, será lo mismo referirse a soluciones ineficientes o exponenciales, si bien puede establecerse una «jerarquía de ineficiencia» dentro de (R – P). De ahora en más, en este marco, R será entonces la clase temporal EXP (clase de problemas con solución determinísica exponencial).
Ahora queda claro por qué es razonable utilizar las MTD con K cintas como modelo estándar en el contexto de la complejidad temporal: si un problema tiene solución polinomial mediante una MTD con K_1 cintas, entonces también tiene solución polinomial mediante una MTD con K_2 cintas, cualesquiera sean K_1 y K_2. En cambio, el modelo de MTN no sería razonable: se probó antes que pasar de una MTN a una MTD puede aumentar el tiempo en el orden exponencial, es decir $O(d^n)$, siendo d una constante que depende de la MTN, por lo que podría darse la inconsistencia de que un problema con solución exponencial mediante una MTD tuviera solución polinomial mediante una MTN.
Además de utilizar un modelo razonable de MT, también se debe elegir una representación razonable de los inputs, porque de lo contrario se producirían inconsistencias como las que muestra el ejemplo siguiente:

Ejemplo. Sea L un conjunto de números naturales, reconocido por una MTD M tal que dado el input w, M trabaja en tiempo T(w) = O(w). Por ejemplo, éste podría ser el caso de un algoritmo muy elemental para determinar si un número w es primo: se chequea si alguno de los números 2, 3, 4, …, w – 1, divide a w; si se cumple se rechaza, y si no se cumple se acepta. Para facilitar la presentación, se asume que el tiempo que lleva cada división es constante.

Si w se representa en notación unaria, entonces |w| = w = n, y por lo tanto M trabaja en tiempo O(w) = O(n), es decir en tiempo lineal con respecto a la longitud de los inputs. Si en cambio w se representa en notación binaria, entonces $|w| = \log_2 w = n$, y por lo tanto se cumple que M trabaja en tiempo $O(w) = O(2^n)$, es decir en tiempo exponencial con respecto a la longitud de los inputs.

Obviamente deben evitarse inconsistencias como las de este ejemplo. Si un problema es tratable con una representación r_1, también lo debe ser con una representación r_2, y si es intratable con r_1 debe serlo con r_2. Esta es la noción de *representación razonable* que se utilizará en lo que sigue. Las características básicas serán:

- *Los números se representan en notación distinta de la unaria*. Como se vio en el ejemplo, puede darse que un problema con solución exponencial utilizando notación no unaria tenga solución polinomial con notación unaria. En cambio, esto no sucede entre las notaciones no unarias, cualesquiera sean las bases.
- *Transformar una representación razonable en otra representación razonable tarda tiempo determinístico polinomial*. Notar en particular, que de la notación no unaria a la notación unaria se pasa en tiempo exponencial, mientras que la relación entre las longitudes de dos números representados en notación no unaria, por ejemplo en base *a* y en base b, es una constante. Más precisamente, el número w en notación base a mide $\log_a w$, en notación base b mide $\log_b w$, y se cumple que $(\log_a w / \log_b w) = \log_a b$. De este modo, en general, se podrá obviar la mención explícita de la base del logaritmo.
- *La representación de los conjuntos consiste en la secuencia de sus elementos, separados por símbolos apropiados*.

Todavía hay una tercera condición de razonabilidad, que se refiere a la elección de los elementos que representan las instancias de los problemas. El ejemplo siguiente muestra que también en este aspecto puede haber inconsistencias.

Ejemplo. Sea L = {G | G es un grafo y tiene un camino del vértice v_1 al vértice v_2 de longitud al menos K}.

El lenguaje L representa el problema de determinar si un grafo tiene un camino entre sus vértices v_1 y v_2 de longitud al menos K. Una representación habitual de un grafo G, que se utilizará en el libro, es la siguiente:

$$G = (V, E),$$

donde V es el conjunto de vértices y E es el conjunto de arcos de G. Los vértices se representan por números naturales en notación binaria y se separan por «#». Los arcos se representan por pares de números naturales en binario y se separan por «#», lo mismo que los números de cada par. Finalmente, V y E se separan por «##».

Con esta representación, no se conoce solución determinística polinomial para L. Tampoco si G se representa por una matriz de adyacencia, en que en las filas y columnas se listan los vértices, y los cruces tienen marcas sii los vértices correspondientes determinan un arco. Y lo mismo sucede con otras representaciones de grafos. En todos los casos, el pasaje de una representación a otra es eficiente.
En cambio, si G se representa enumerando todos sus caminos, una solución eficiente trivial para L es la siguiente: se recorren todos los caminos, y se busca un camino entre v_1 y v_2 de longitud al menos K. Esta representación no es razonable, la complejidad temporal del problema está oculta en su representación. Además, el tiempo de pasaje de cualquiera de las representaciones mencionadas antes a ésta es exponencial.

Las características de las clases P y NP se tratarán en detalle en el próximo capítulo. En lo que sigue, se presentan características generales de la jerarquía temporal. Si bien sólo se considera el tiempo determinístico, debe entenderse que todo aplica también al tiempo no determinístico.
En primer lugar, debe notarse que aunque se cumpla que $T_1(n) = O(T_2(n))$, dadas dos funciones T_1 y T_2, la clase $DTIME(T_2(n))$ no necesariamente tendrá más lenguajes que la clase $DTIME(T_1(n))$. Por ejemplo, el «salto» de una clase a otra debe ser mayor que lo determinado por un factor constante:

Teorema de *Linear Speed Up* (aceleración lineal). Si existe una MTD M_1 que trabaja en tiempo $cT(n)$, con $c > 0$, entonces existe una MTD M_2 equivalente que trabaja en tiempo $T(n)$.

Prueba. Sólo se presenta la idea general. Si M_1 trabaja en tiempo determinístico cT(n), entonces a partir de cualquier input w tal que |w| = n, M_1 hace a lo sumo cT(n) pasos. La idea es construir una MTD M_2 equivalente a M_1 tal que sus alfabetos Σ_2 y Γ_2 tengan como símbolos secuencias de k símbolos de los alfabetos Σ_1 y Γ_1, para un k determinado que depende de c (es decir que las cintas de M_2 tendrán k tracks). El propósito es que lo que a la MT M_1 le lleva c pasos, a la MT M_2 le lleve sólo un paso, definiendo una función de transición apropiada.

□

Se prueba que recién cuando $\inf_{n \to \infty} (T_1(n) \log_2 T_1(n) / T_2(n)) = 0$, y $T_2(n)$ «se comporta bien», entonces $DTIME(T_1(n)) \neq DTIME(T_2(n))$. En particular, si además se cumple que $T_1(n) = O(T_2(n))$, entonces vale que $DTIME(T_1(n)) \subset DTIME(T_2(n))$.

En otras palabras, se pasa de una clase temporal $DTIME(T_1(n))$ a una clase temporal $DTIME(T_2(n))$ que la incluye estrictamente, si $T_2(n)$ es significativamente mayor que $T_1(n) \log_2 T_1(n)$ cuando n tiende a infinito, y $T_2(n)$ «se comporta bien», lo que significa formalmente que T_2 es una función *tiempo-construible*:

Definición. Una función $T: N \to N^+$ es tiempo-construible, si existe una MTD que para todo input w, tal que |w| = n, trabaja en tiempo exactamente T(n).

□

La condición de una función de ser tiempo-construible, es más fuerte que la condición de ser total computable. Si T(n) es tiempo-construible, entonces dado cualquier n, T(n) no sólo es computable sino que además se computa en tiempo exactamente T(n). Queda como ejercicio para el lector, probar que una función tiempo-construible es también una función total computable.
Las funciones tiempo-construibles se utilizan como «relojes» en las simulaciones, cuando una MT debe simular otra exactamente un número de pasos.
La referencia al «buen comportamiento» que aparece antes, tiene que ver con que cuando se trata con funciones que no son tiempo-construibles, pueden producirse efectos no deseados en la jerarquía temporal. Por ejemplo, dada una MT siempre podría existir otra más eficiente (esto se formula por un teorema denominado Teorema de *Speed Up*). Otro caso es que aún aumentando sobremanera una función

T(n), podría no obtenerse una clase temporal mayor (lo que se formula mediante el Teorema del *Gap*).
Afortunadamente, todas las funciones con las que se trabaja habitualmente en la complejidad temporal, tales como n^k, 2^n, n!, etc., son tiempo-construibles. Al final de esta parte del libro se formulan algunos ejercicios relacionados con este tema.
En base a estas consideraciones, por ejemplo para el caso de las funciones de la forma n^k, se cumple que DTIME(n^k) $\subset$ DTIME(n^{k+1}):

Ejemplo. Sean las clases temporales DTIME(n^3) y DTIME(n^4). Se cumple DTIME(n^3) $\subset$ DTIME(n^4):

- La función $T(n) = n^4$ es tiempo-construible.
- Se cumple que: $\inf_{n \to \infty} (n^3 \log_2 n^3 / n^4) = 0$.
- Se cumple que: $n^3 = O(n^4)$.

Lo anterior se puede generalizar para todo par de clases DTIME(n^k) y DTIME(n^{k+1}) del conjunto P.

Por lo tanto, dada una clase temporal DTIME($T_1(n)$), siempre se puede encontrar otra clase DTIME($T_2(n)$) que la incluya estrictamente. En este sentido, la jerarquía temporal dentro de R es «densa». Además, evitando la utilización de funciones que no sean tiempo-construibles (con lo que no se pierde generalidad como se indicó antes), la jerarquía no tiene *gaps*.
El siguiente teorema refuerza esta característica de la jerarquía temporal, aún sin recurrir al uso de las funciones tiempo-construibles:

Teorema. Si $T: N \to N^+$ es una función total computable, entonces existe un lenguaje recursivo L tal que $L \notin$ DTIME(T(n)).

Prueba. Se usará diagonalización, para encontrar un lenguaje L por fuera de DTIME(T(n)). Sea el lenguaje $L = \{x_i \mid x_i$ no es aceptado por la MT M_i en $T(|x_i|)$ pasos$\}$, siendo T una función total computable, y el índice i de x_i y M_i el correspondiente al orden canónico.

Se cumple que $L \varepsilon R$. La siguiente MT M reconoce L y se detiene siempre: dado el input w, M calcula i tal que $w = x_i$, genera $<M_i>$, calcula k =

T(|w|), y simula k pasos de M_i a partir de w, aceptando sii M_i no acepta w. Es fácil comprobar que L(M) = L y que M se detiene siempre.
Se cumple además que $L \notin DTIME(T(n))$. Supóngase que $L \in DTIME(T(n))$. Sea M_i una MT que reconoce L en tiempo T(n) (se puede obviar la constante c por el Teorema de *Linear Speed Up*):

a) Si $x_i \in L$, entonces M_i acepta x_i en tiempo $T(|x_i|)$, pero entonces por la definición de L, se cumple que $x_i \notin L$ (absurdo).
b) Si $x_i \notin L$, entonces M_i no acepta x_i, y en particular no lo acepta en tiempo $T(|x_i|)$, pero entonces por la definición de L, se cumple que $x_i \in L$ (absurdo).

De acuerdo a este teorema, entonces otra manera de obtener una clase que incluya estrictamente a DTIME(T(n)) es la que se presenta en el siguiente corolario:

Corolario. Sea la clase temporal $DTIME(T_1(n))$, la función T_1 que es una función total computable, y el lenguaje L del conjunto $(R - DTIME(T_1(n)))$. Supóngase que $L \in DTIME(T_2(n))$.

- Si $T_1(n) = O(T_2(n))$, entonces $DTIME(T_1(n)) \subset DTIME(T_2(n))$.
- Si no, sea la función T_3, tal que $\forall n \in N$, $T_3(n) = \max(T_1(n), T_2(n))$, siendo $\max(n_1, n_2)$ el máximo entre n_1 y n_2. Se cumple que $DTIME(T_1(n)) \subset DTIME(T_3(n))$.

□

En el Apéndice 2 se describen resumidamente otras jerarquías analizadas en la complejidad computacional, basadas en modelos de computación alternativos al de las máquinas de Turing. Estableciendo relaciones entre las jerarquías, se facilita el entendimiento de las propiedades de las distintas clases de problemas.
Con el mismo propósito, también en el Apéndice 2 se presentan suscintamente problemas más generales que los de decisión y se los relaciona con los de decisión, y en los capítulos de esta parte del libro dedicados a la complejidad espacial, se describen relaciones entre las jerarquías espacial y temporal.

CAPÍTULO 2.2. LA CLASE NP

En este capítulo se analiza en detalle la clase temporal NP, la cual nuclea importantes problemas de la ciencia de la computación y las matemáticas.
En la Sección 2.2.1 se presentan las características principales de la clase NP. Se describe cómo se prueba que un problema pertence a NP por medio de la construcción de una máquina de Turing no determinística. Y se establece una importante caracterización de las clases P y NP: respectivamente, se puede comparar la dificultad para probar la pertenencia a ellas, con la dificultad para construir la prueba de un teorema con respecto a la dificultad para verificar la prueba de un teorema.
En la sección 2.2.2 se describe la estructura interna de la clase NP. Particularmente, se destacan los problemas NP-completos, que por medio de la técnica de las reducciones polinomiales (reducciones que se ejecutan en tiempo determinístico polinomial), se prueba que son los más complejos de NP.
La NP-completitud es uno de los conceptos más relevantes de la teoría de la complejidad computacional. Es que asumiendo la conjetura $P \subset NP$, probar que un problema es NP-completo significa, en la práctica, que es intratable. Lo interesante es que importantes problemas de la ciencia de la computación y las matemáticas pertencen a esta clase. Se presenta el Teorema de Cook, que introduce un primer problema NP-completo, el problema de decidir si una fórmula booleana es satisfactible. Por medio de las reducciones polinomiales, a partir de este primer problema, se pueden obtener nuevos problemas NP-completos.
También se describe, dentro de NP, la clase de los problemas de complejidad temporal intermedia. Otra vez, asumiendo la conjetura $P \subset NP$, se demuestra que esta clase no es vacía.
En la Sección 2.2.3 se hace una breve referencia al exterior de la clase NP. Se considera el conjunto de los lenguajes tales que sus complementos pertenecen a NP, es decir la clase CO-NP de lenguajes (o problemas). También se describen características generales de los problemas con resolución temporal exponencial. Se mencionan problemas con cota mínima temporal exponencial.

2.2.1. Características generales de la clase NP

Ya se han presentado algunos ejemplos de lenguajes de la clase temporal P. En cada caso, se probó la existencia de una MTD que reconoce el lenguaje y que trabaja en tiempo $O(n^k)$, con k constante.
Una manera alternativa para probar que $L \varepsilon P$ es probar que su complemento $L^C \varepsilon P$ (o lo que es lo mismo, que $L \varepsilon$ CO-P, denominando CO-P a la clase de lenguajes recursivos tales que sus complementos están en P). Esto se cumple porque la clase P es cerrada con respecto a la operación de complemento. Formalmente:

Teorema. Si L es un lenguaje de P, entonces también lo es su complemento L^C.

Prueba. Si M es una MTD que reconoce L en tiempo polinomial, entonces la MTD M' que se comporta como M salvo que M' acepta sii M rechaza, reconoce L^C en tiempo polinomial (de hecho M' trabaja en el mismo tiempo que M).

□

Si se probara que NP no es cerrada con respecto al complemento, se estaría probando entonces que $P \neq NP$. Encontrar una propiedad que se cumpla en una clase y no en la otra es un camino viable para probar la desigualdad $P \neq NP$. En cambio, como se mostrará en el Apéndice 2, no sirven para esta prueba ni la técnica de simulación para demostrar P = NP (es decir que para toda MTN que trabaja en tiempo polinomial existe una MTD equivalente que la simula en tiempo polinomial), ni la técnica de diagonalización para demostrar $P \neq NP$ (es decir que se puede diagonalizar para encontrar un lenguaje en el conjunto (NP – P)).
Otra forma de probar la pertenencia a la clase P (y también a la clase NP en general), es utilizando reducciones de problemas, ahora agregándoles un atributo de tiempo. Esta técnica se introducirá más adelante.
La forma natural de probar que un lenguaje pertenece a la clase NP, es construir una MTN que lo reconozca y trabaje en tiempo polinomial. El siguiente es un primer ejemplo de este tipo de prueba.

Ejemplo. Sea HC = {G | G es un grafo que tiene un circuito de Hamilton}.

El lenguaje HC representa el problema de determinar si un grafo tiene un circuito de Hamilton. Se define que un grafo G = (V, E), con m vértices, tiene un circuito de Hamilton C, si C es una permutación $<v_{i1}, v_{i2}, v_{i3}, \ldots, v_{im}>$ de V, tal que los arcos (v_{i1}, v_{i2}), (v_{i2}, v_{i3}), (v_{i3}, v_{i4}), …, (v_{im-1}, v_{im}), (v_{im}, v_{i1}) pertenecen a E. Es decir, un circuito de Hamilton recorre todos los vértices de un grafo sin repetirlos, arrancando y terminando en un mismo vértice.
Por convención, se asumirá de ahora en más que un grafo tiene m vértices. Cuando no sea necesario, no se explicitará si un grafo es o no orientado. G se representará por un par (V, E), como ya se describió en un ejemplo previo, utilizando números en binario para los nombres de los vértices, y el símbolo especial «#» como separador.
No se conoce solución determinística polinomial para este problema. La solución determinística natural, que chequea una a una las permutaciones de V para detectar si una de ellas es un circuito de Hamilton de G, tarda tiempo exponencial: hay m! permutaciones de V, por lo que el tiempo de la solución será en el peor caso, obviando incluso el tiempo de algunos pasos de la solución:

$$O(m!) = O(m^m) = O(m^{|V|}) = O(m^{|G|}) = O(m^n)$$

Se prueba en cambio fácilmente, que HC ε NP. Se va a construir una MTN M que reconoce HC en tiempo polinomial. Dado el input G = (V, E), la MT M hace:

1) Si G es un grafo inválido, rechazar.
2) Generar no determinísticamente un string C, con |C| = |V|.
3) Aceptar sii C es un circuito de Hamilton de G.

Se cumple que HC = L(M):

G ε HC ↔ G tiene un circuito de Hamilton ↔ M genera en el paso (2) de alguna computación un circuito de Hamilton ↔ M acepta G ↔ G ε L(M).

Además, M trabaja en tiempo no determinístico polinomial:

– En (1) hay que verificar que los vértices de V están ordenados y no se repiten, que existe el separador «##», que en E no hay arcos repetidos y que los vértices de los arcos están en V, etc. Considerando estas validaciones, el tiempo sería: O(|V|) + O(1) +

$O(|E|^2) + O(|V||E|) = O(|G|) + O(1) + O(|G|^2) + O(|G|^2) = O(|G|^2) = O(n^2)$.

- En (2) hay que generar no determinísticamente un string C con símbolos de {0, 1, #} de tamaño |V|. Esto tarda: $O(|V|) = O(|G|) = O(n)$.
- Finalmente en (3) hay que chequear que $C = <v_{i1}, v_{i2}, v_{i3}, \ldots, v_{im}>$ es una permutación de V, y que los arcos (v_{i1}, v_{i2}), (v_{i2}, v_{i3}), (v_{i3}, v_{i4}), …, (v_{im-1}, v_{im}), (v_{im}, v_{i1}) están en E. Esto tarda: $O(|V|^2) + O(|V||E|) = O(|G|^2) + O(|G|^2) = O(|G|^2) = O(n^2)$.

Por lo tanto, la MTN M tarda: $O(n^2) + O(n) + O(n^2) = O(n^2)$.

Obviamente que si se demostrara que HC $\notin$ P, entonces se probaría que la clase NP incluye estrictamente a la clase P.

La forma de la MTN que trabaja en tiempo polinomial, construida para demostrar que un lenguaje está en NP, será siempre la misma:

a) Se genera no determinísticamente, en tiempo polinomial, una posible solución del problema.
b) Se chequea, en tiempo polinomial, si lo generado en (a) es, en efecto, una solución del problema. El chequeo incluye la validación sintáctica del input, que por razones de eficiencia es lo que primero se ejecuta.

Enseguida se va a presentar una definición equivalente de la clase NP, relacionada con esta última consideración. La figura 2.2.1 ilustra la forma de la MTN referida.

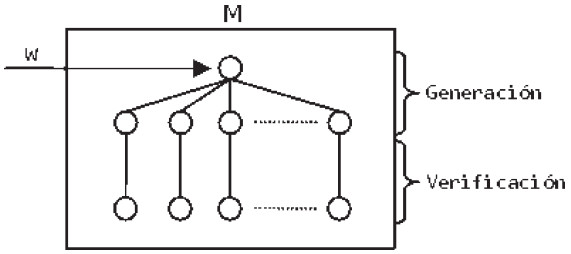


Figura 2.2.1

Se puede hacer un paralelismo entre las pruebas de pertenencia a P y NP, y las pruebas de los teoremas: la dificultad para probar que un lenguaje está en P se puede equiparar con la dificultad para construir la prueba de un teorema, mientras que la dificultad para probar que un lenguaje está en NP se puede equiparar con la dificultad para verificar la prueba de un teorema.
El siguiente es otro ejemplo de prueba de pertenencia a NP por construcción de una MTN:

Ejemplo. Sea CLIQUE = {(G, K) | G es un grafo que tiene un clique de tamaño K}.

El lenguaje CLIQUE representa el problema de determinar si un grafo tiene un clique de tamaño K. Un clique de tamaño K de un grafo G es un subgrafo completo de G con K nodos. También se lo puede definir como un conjunto de vértices $Y = \{v_{i1}, v_{i2}, v_{i3}, \ldots, v_{iK}\} \subseteq V$, tal que para todo par de nodos v, v' de Y, se cumple que $(v, v') \varepsilon E$.
No se conoce solución determinística polinomial para el problema. La solución determinística natural consiste en chequear uno a uno los subconjuntos de K vértices de V para detectar si uno de ellos es un clique de G. Esta validación tarda tiempo exponencial, porque existen $m! / ((m – K)! K!)$ subconjuntos de K vértices en V, y así, el tiempo de trabajo será en el peor caso, obviando incluso algunos pasos de la solución:

$$O(m(m – 1)(m – 2)\ldots(m – K + 1) / K!) = O(m^K) = O(m^n).$$

Se prueba fácilmente, en cambio, que CLIQUE se puede reconocer en tiempo no determinístico polinomial, es decir que CLIQUE ε NP. La siguiente MTN M reconoce CLIQUE en tiempo polinomial. Dado el input w = (G, K), con G = (V, E), la MTN M hace:

1) Si el input no es válido, rechazar (por ejemplo, G no es un grafo válido, o no existe un separador «##» entre G y K, ó K > m, etc).
2) Generar no determinísticamente un string Y, con $|Y| \leq |V|$.
3) Aceptar sii Y es un clique de G de tamaño K.

Se cumple que CLIQUE = L(M):

(G, K) ε CLIQUE $\leftrightarrow$ G tiene un clique de tamaño K $\leftrightarrow$ M genera en el paso (2) de alguna computación un clique de G de tamaño K $\leftrightarrow$ M acepta (G, K) $\leftrightarrow$ (G, K) ε L(M).

Se cumple, además, que la MT M trabaja en tiempo no determinístico polinomial:

- En (1) hay que verificar la sintaxis del input, lo que tarda $O(|G|^2) = O(n^2)$.
- En (2) hay que generar no determinísticamente un string Y con símbolos de {0, 1, #} de tamaño a lo sumo |V|, lo que tarda $O(|V|) = O(|G|) = O(n)$.
- Finalmente, en (3) hay que verificar que Y tiene K vértices distintos de G, y que todo par de vértices son adyacentes. Esto tarda: $O(|V|^2) + O(|V|^2|E|) = O(|G|^2) + O(|G|^3) = O(|G|^3) = O(n^3)$.

Por lo tanto, la MTN M tarda: $O(n^2) + O(n) + O(n^3) = O(n^3)$.

Es interesante remarcar, siguiendo con el ejemplo anterior, que si K no formara parte del input, entonces el lenguaje estaría en P. Denominando KCLIQUE al nuevo lenguaje, sería:

KCLIQUE = {G | G es un grafo que tiene un clique de tamaño K}, con K constante.

En este caso, recorrer determinísticamente m! / ((m – K)! K!) subconjuntos de K vértices en V, tardaría:

$$O(m(m – 1)(m – 2)\ldots(m – K + 1) / K!) = O(m^K) = O(n^K),$$

con K constante, es decir un tiempo del orden polinómico con respecto a la longitud del input. De todos modos podría discutirse si para un valor de K muy grande, por ejemplo 1000, el tiempo de la MTD construida debe considerarse eficiente.

Por otro lado, esta otra variante del lenguaje anterior no parece pertenecer siquiera a NP:

NOCLIQUE = {(G, K) | G es un grafo que no tiene un clique de tamaño K}. Una MTN M que reconozca este lenguaje, debería recorrer en cada una de sus computaciones todos los subconjuntos de K vértices de V,

para aceptar o rechazar adecuadamente, con lo que el tiempo de M sería exponencial. No se conoce solución no determinística polinomial para el lenguaje NOCLIQUE.

***Ejercicio**. Sea M' una MTN idéntica a la MTN M que se construyó antes para probar que el lenguaje CLIQUE ε NP, salvo que M' rechaza sii M acepta. Explicar por qué L(M') $\neq$ NOCLIQUE.*

□

Para cerrar esta sección de generalidades de la clase NP, se presenta una definición equivalente de la misma, referida previamente.
Informalmente, y manteniendo la visión de que los elementos de un lenguaje representan las instancias de un problema, la idea es que todo elemento x de un lenguaje L ε NP, tiene una solución *y* de tamaño polinomial con respecto al tamaño de x, que se verifica en tiempo determinístico polinomial con respecto al tamaño de x. Esto se comprobó en los ejemplos anteriores (problema del circuito de Hamilton y problema del clique de tamaño K).

El siguiente teorema formaliza la definición alternativa de la clase NP:

Teorema. L $\in$ NP $\leftrightarrow$ existe un predicado de dos argumentos R ε P y un polinomio Q, tales que L = {x | $\exists$y: (|y| $\leq$ Q(|x|)) $\wedge$ R(x, y)}.

Se define que un predicado R pertenece a la clase P, sii existe una MTD que computa R (es decir que acepta o rechaza adecuadamente los inputs) en tiempo polinomial.
También se utilizará después la noción de una función f perteneciente a la clase P (podría usarse también la denominación FP). Se define que f ε P, sii existe una MTD que computa f en tiempo polinomial, utilizando una cinta de output de solo escritura sobre la que el cabezal se mueve únicamente a la derecha.

Prueba. La prueba de que si existe un predicado de dos argumentos R ε P y un polinomio Q tales que L = {x | $\exists$y: (|y| $\leq$ Q(|x|)) $\wedge$ R(x, y)}, entonces L ε NP, es trivial y queda como ejercicio para el lector.
Se probará el sentido contrario. Es decir, se probará que dado L $\in$ NP, existe un predicado de dos argumentos R ε P y un polinomio Q, tales que L = {x | $\exists$y: (|y| $\leq$ Q(|x|)) $\wedge$ R(x, y)}.

Sea M_1 una MTN que reconoce L en tiempo polinomial. Se prueba fácilmente que existe una MTN M_2 equivalente a M_1 que trabaja en tiempo polinomial y cuya relación de transición Δ tiene grado 2 (la demostración queda como ejercicio para el lector). Se va a construir una MTN M_3 equivalente a M_2 que trabaja también en tiempo polinomial. Suponiendo que M_2 trabaja en tiempo polinomial Q(n) con inputs de tamaño n, y que su relación de transición se denomina Δ_2, sea la siguiente MTN M_3, que a partir del input w hace:

1) Calcular Q(|w|), y generar no determinísticamente un string *y* de {0, 1}* de longitud Q(|w|).
2) Computar determinísticamente el predicado R(w, y), que consiste en: (a) Simular M_2 a partir de w, de acuerdo a la secuencia *y* de ceros y unos generada. Por ejemplo, si *y* empieza con «0110», simular el primer paso de M_2 tomando la primera alternativa de Δ_2, luego simular el segundo paso de M_2 tomando la segunda alternativa de Δ_2, luego otra vez la segunda alternativa, luego la primera, etc. Finalmente aceptar sii M_2 acepta.

Se cumple que $L(M_2) = L(M_3)$:
$w \varepsilon L(M_2) \leftrightarrow$ existe una computación de M_2 que acepta $w \leftrightarrow$ por construcción de M_3 a partir de M_2, existe una computación de M_3 que acepta $w \leftrightarrow w \varepsilon L(M_3)$.

Además, la MTN M_3 trabaja en tiempo polinomial, porque el paso (1) tarda tiempo no determinístico O(Q(|w|)), y el paso (2) tarda tiempo determinístico O(Q(|w|)).

De este modo, mediante la construcción de la MTN M_3, se mostró que hay un predicado de dos argumentos $R \varepsilon P$ y un polinomio Q, tales que para todo elemento $w \varepsilon L$, existe un string *y* que cumple $|y| \leq Q(|w|)$ y R(w, y).

□

2.2.2. El interior de la clase NP

Para caracterizar las distintas clases de problemas que componen NP, será sumamente importante la utilización de las reducciones polinomiales, que se definen a continuación.

2.2.2.1. Reducciones polinomiales

Definición. Sean L_1 y L_2 dos lenguajes incluidos en Σ^*. Existe una *reducción polinomial* de L_1 a L_2 sii existe una reducción de L_1 a L_2 y la función de reducción f pertenece a la clase P.

□

Se va a utilizar la expresión:

$$L_1 \alpha_P L_2,$$

para denotar que existe una reducción polinomial de L_1 a L_2. Como antes, M_f identifica la MT M que computa la función de reducción f (en este caso en tiempo determinístico polinomial), y f_M es la función de reducción f computada por dicha MT.
El siguiente teorema fundamenta cómo se van a utilizar las reducciones polinomiales para «poblar» las clases P y NP.

Teorema. Si $L_1 \alpha_P L_2$, entonces: (a) $L_2 \varepsilon P \rightarrow L_1 \varepsilon P$, y (b) $L_2 \varepsilon NP \rightarrow L_1 \varepsilon NP$.

Prueba. Se probará sólo el caso (a). El caso (b) queda como ejercicio para el lector.

La idea general de la prueba se ilustra en la figura 2.2.2.

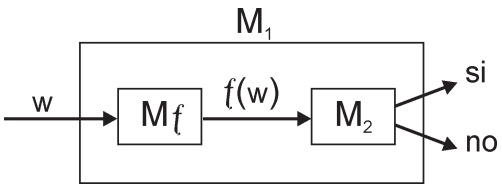


Figura 2.2.2

La MTD M_1 compone M_f con M_2, siendo M_f la MTD que reduce el lenguaje L_1 al lenguaje L_2 en tiempo polinomial, y M_2 la MTD que reconoce L_2 en tiempo polinomial. M_1 reconoce L_1, tal como se probó en la primera parte del libro, y trabaja en tiempo polinomial determinístico porque así lo hacen M_f y M_2. De esta manera, se cumple que $L_1 \varepsilon P$. Formalmente:

– si M_f es la MTD que computa la función de reducción f entre L_1 y L_2 en a lo sumo $a|w|^b$ pasos, para todo input $w \varepsilon \Sigma^*$, siendo a, b, constantes,

– si M_2 es la MTD que reconoce L_2 en a lo sumo $c|w|^d$ pasos, para todo input $w \varepsilon \Sigma^*$, siendo c, d, constantes,

– y si M_1 es una MTD que, dado un input w, primero simula M_f a partir de w para obtener f(w), y luego simula M_2 a partir de f(w) aceptando sii M_2 acepta,

– entonces $L_1 \varepsilon P$, porque M_1 reconoce el lenguaje L_1 en tiempo determinístico polinomial:

  a) M_1 reconoce L_1: esto se probó en la primera parte del libro para el caso general de dos lenguajes recursivos L_1 y L_2 .
  b) M_1 trabaja en tiempo determinístico polinomial: dado un input w, M_1 primero hace a lo sumo $a|w|^b$ pasos al simular M_f para obtener f(w), con $|f(w)| \leq a|w|^b$ (porque en k pasos no puede generarse un string de más de k símbolos); M_1 después hace a lo sumo $c(a|w|^b)^d$ pasos al simular M_2 a partir de f(w); de esta manera, en total, M_1 hace a lo sumo $(a|w|^b + c(a|w|^b)^d)$ pasos, es decir $O(|w|^e)$ pasos, siendo e alguna constante.

□

Por lo tanto, se prueba que un lenguaje L está en P (respectivamente NP), si se lo puede reducir polinomialmente a otro lenguaje L' de P (respectivamente NP).
Se demuestra fácilmente que la relación α_P es reflexiva y transitiva. No es simétrica porque ni siquiera la relación α lo es: se puede reducir cualquier lenguaje recursivo L al lenguaje L_U (esta reducción se presenta más adelante), pero no se puede reducir L_U a L porque de lo contrario el lenguaje L_U sería recursivo.

Ejercicio. *Probar que la relación α_P es reflexiva y transitiva.*

□

El siguiente ejemplo de reducción polinomial se refiere a un problema clásico de la literatura, el problema del viajante de comercio.

Ejemplo. Sea el lenguaje TSP = {(G, B) | G es un grafo completo, los arcos tienen asociado un costo, y G tiene un circuito de Hamilton tal que la suma de los costos de sus arcos es menor o igual que B}.
El lenguaje TSP (por *travelling salesman problem*) representa el problema del viajante de comercio. El viajante de comercio debe visitar un conjunto de ciudades una sola vez, y volver al punto de partida, de manera tal de no recorrer más que una determinada distancia.
Para representar los costos (longitudes) de los arcos, después de cada arco se va a codificar un número natural en binario, separándolos por el símbolo «#».

Se cumple que: HC α_P TSP. Se propone una función de reducción f tal que f(G) = (G', m), con:

- G' es un grafo completo con los mismos vértices de G. Si (i, k) es un arco de G, entonces su costo en G' es 1, y si (i, k) no es un arco de G, entonces su costo en G' es 2.
- El número m, como siempre, es el número de vértices de G' (y de G).

La figura 2.2.3 ilustra la reducción planteada.

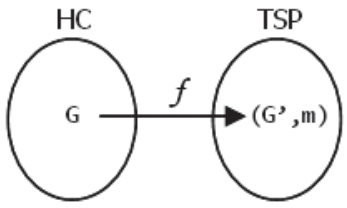


Figura 2.2.3

Se cumple que f es una función total computable:

Existe una MT M_f que a partir de G, primero chequea si G es válido sintácticamente, y si no lo es genera como output el string «1» que no es un par válido (G', m). En caso contrario, M_f copia los vértices de G, copia los arcos de G y les asigna el costo 1, agrega los arcos que G no tiene y les asigna el costo 2, y finalmente escribe el número m.

Se cumple que G ε HC $\leftrightarrow$ (G', m) ε TSP:

G ε HC $\leftrightarrow$ G tiene un circuito de Hamilton $\leftrightarrow$ por construcción, G' tiene un circuito de Hamilton de longitud m $\leftrightarrow$ (G', m) ε TSP.

Finalmente, también se cumple que f ε P:

M_f es una MTD que trabaja en tiempo polinomial, más precisamente $O(n^3)$, porque: (a) chequear la validez sintáctica de G tarda $O(n^2)$; (b) escribir el string «1» tarda O(1); (c) escribir V tarda O(n); (d) escribir E agregando el número 1 como costo a cada arco tarda O(n); (e) escribir los arcos que no tiene G agregando el número 2 como costo a cada arco tarda $O(n^3)$; y (f) escribir el número m tarda O(n).

Se prueba fácilmente, por construcción de una MTN, que TSP ε NP. Por lo tanto, como se cumple HC α_P TSP, en base al teorema anterior se podría demostrar que HC ε NP sin pasar por la construcción de una MTN que reconozca HC en tiempo polinomial.
No se conoce solución determinística polinomial para TSP. Si existiera, por el teorema anterior valdría que HC ε P. Generalizando, si existe una reducción polinomial de L_1 a L_2, entonces desde el punto de vista de la complejidad temporal se cumple que el lenguaje L_2 es tan o más difícil que el lenguaje L_1. Volviendo al ejemplo, el problema del viajante de comercio es tan o más difícil de resolver que el problema del circuito de Hamilton, en cuanto al tiempo de resolución.
Ya definidas las reducciones polinomiales, en la siguiente sección se las va a utilizar para estudiar la estructura interna de NP. En base a lo demostrado previamente, podrá establecerse una jerarquía temporal más detallada dentro de la clase.
Para estudiar la estructura interna de P, en cambio, deberá recurrirse a otro tipo de reducciones, lo que se verá cuando se trate la complejidad espacial (también se hará otra mención en el Apéndice 2). Esto se debe a que siempre se puede reducir polinomialmente de un lenguaje de P a otro lenguaje de P. Formalmente, sean L_1 y L_2 dos lenguajes de P, $L_2 \neq \Sigma^*$ y $L_2 \neq \emptyset$. Se cumple que L_1 α_P L_2: si $a \varepsilon L_2$, y $b \notin L_2$, entonces la siguiente MTD M_f computa una reducción polinomial de L_1 a L_2: dado w, M_f primero decide en tiempo polinomial si w ε L_1 ó w ε $(\Sigma^* - L_1)$, y luego escribe (en tiempo constante) a ó b, respectivamente.

2.2.2.2. Problemas NP-completos

Asumiendo P ≠ NP, mediante el concepto de NP-completitud podrá «probarse» la intratabilidad de determinados problemas de NP.

La clase NP va a quedar dividida en, por un lado los lenguajes de P, representantes de problemas con solución eficiente, y por el otro los lenguajes NP-completos, representantes de problemas con solución ineficiente. Y además entre ellos aparecerá un tercer conjunto de lenguajes, de dificultad intermedia.
Las siguientes definiciones introducen el concepto de NP-completitud:

Definición. Un lenguaje L_0 es *NP-difícil* (o *NP-hard*) sii para todo lenguaje $L \varepsilon NP$ se cumple que $L \alpha_P L_0$. Es decir, L_0 es un lenguaje NP-difícil sii todos los lenguajes de NP se reducen polinomialmente a él.

□

La clase de los lenguajes NP-difíciles se va a denominar NPH (ver la figura 2.2.4).

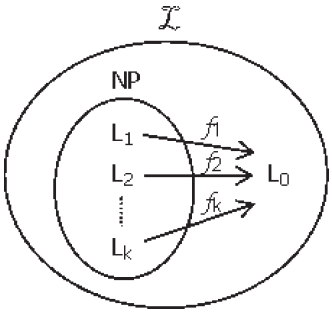


Figura 2.2.4

Si un lenguaje NP-difícil pertenece a la clase NP, se define que es un lenguaje NP-completo. Formalmente:

Definición. Un lenguaje L_0 es *NP-completo* sii $L_0 \varepsilon$ (NP $\cap$ NPH). Es decir, L_0 es un lenguaje NP-completo sii está en NP y todos los lenguajes de NP se reducen polinomialmente a él.

□

La clase de los lenguajes NP-completos se va a denominar NPC (ver la figura 2.2.5).

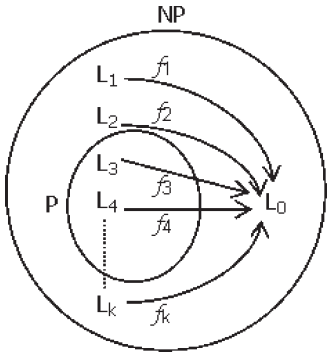


Figura 2.2.5

El siguiente teorema formaliza lo que se anticipó al comienzo de esta sección, en cuanto a la utilidad de la noción de NP-completitud para determinar que un problema de NP es intratable.

Teorema. Sea L un lenguaje NP-completo. Si L ε P, entonces P = NP.

En otras palabras, asumiendo que P $\neq$ NP, entonces probar que un lenguaje es NP-completo significa «condenarlo» a pertenecer al conjunto (NP – P).

Prueba. Sea L ε NPC y L ε P. Sea L' ε NP. Por ser L un lenguaje NP-completo, entonces se cumple que L' α_p L. Y por ser L un lenguaje de P, entonces por un teorema anterior se cumple que L' ε P. Como esto vale para todo lenguaje L' ε NP, entonces se cumple que NP $\subseteq$ P, y así: P = NP.

□

La figura 2.2.6 ilustra una primera versión de la estructura interna de NP, asumiendo $P \neq NP$.

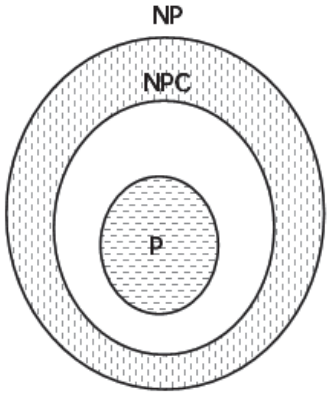


Figura 2.2.6

Como se aprecia, los problemas de NPC están en la franja de NP más alejada de P. Entre la franja de NPC y el círculo central que representa P, hay una franja intermedia de problemas (en blanco, en la figura 2.2.6), a la que se hará referencia más adelante.
El conjunto NPC incluye importantes problemas de la teoría de grafos, la lógica, la aritmética, etc., de los que no se conocen soluciones determinísticas polinomiales. Por otra parte, como lo demuestra el siguiente ejemplo, el conjunto NPH por sí solo no es de mucha utilidad en la práctica, en el marco de la complejidad temporal:

Ejemplo. Se cumple que $L_U \varepsilon$ NPH.

De este modo, un lenguaje NP-difícil puede ser no recursivo.

Prueba. Sea L algún lenguaje de NP. Se va a probar que $L \alpha_P L_U$. Se propone una función de reducción f tal que $f(w) = (<M_L>, w)$, siendo M_L una MTN que reconoce L en tiempo polinomial (ver la figura 2.2.7).

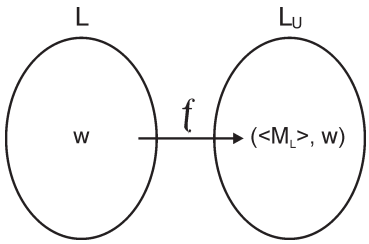


Figura 2.2.7

La función f es una función total computable:

Existe una MT M_f que, a partir de un input w, escribe el código de la MTN M_L seguido de un separador y del string w.

Vale que $w \varepsilon L \leftrightarrow (<M_L>, w) \varepsilon L_U$:

$w \varepsilon L \leftrightarrow M_L$ acepta $w \leftrightarrow (<M_L>, w) \varepsilon L_U$.

Finalmente, también se cumple que $f \varepsilon P$:

M_f es una MTD que trabaja en tiempo polinomial, porque escribir el código $<M_L>$ seguido del separador tarda tiempo O(1), y escribir el string w tarda tiempo O(n), es decir que en total M_f trabaja en tiempo O(n).

□

Se prueba a continuación que NPC ≠ ∅, mediante el Teorema de Cook. Se va a encontrar un primer lenguaje NP-completo, mediante una reducción polinomial de un lenguaje genérico de NP a un lenguaje particular de NP, el lenguaje SAT, que representa el problema de la satisfactibilidad de las fórmulas booleanas.
Una vez probado que SAT es NP-completo, se irá «poblando» NPC por medio de reducciones polinomiales a partir de SAT, como se indicará después. El mecanismo es similar al que se describió en la primera parte del libro, para encontrar lenguajes no recursivos y no recursivamente numerables.

Teorema de Cook. El lenguaje SAT = {ϑ | ϑ es una fórmula booleana satisfactible} es un lenguaje NP-completo.

Prueba. Hay que probar: (a) SAT ε NP, y (b) SAT es NP-difícil.

Parte (a). SAT ε NP:

Una MTN M que, dada la fórmula booleana ϑ,

1) rechaza si ϑ no es una válida,
2) genera no determinísticamente una asignación A de valores de verdad para las m variables de ϑ, y
3) acepta sii A satisface ϑ,

reconoce el lenguaje SAT en tiempo polinomial (queda como ejercicio para el lector probar que los tres pasos anteriores tardan tiempo polinomial).
Notar que la solución determinística natural del problema, consiste en probar en el peor caso 2^m asignaciones de valores de verdad, y por lo tanto tarda a lo sumo $O(2^n)$, con $n = |\vartheta|$.

Parte (b). SAT es NP-difícil:

Se probará que todo lenguaje L ε NP se reduce polinomialmente al lenguaje SAT (ver la figura 2.2.8):

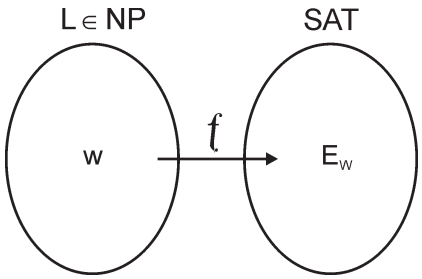


Figura 2.2.8

En la figura 2.2.8, L es algún lenguaje de NP, y E_w es una fórmula booleana. Si M es una MTN que reconoce L en tiempo polinomial p(n), la idea es que M_f transforme todo string w en una fórmula booleana E_w, de modo tal que si alguna computación de M, a partir de w, se detiene en q_A, entonces existe alguna asignación de valores de verdad que satisface E_w, y si en cambio todas las computaciones de M, a partir de w, se detienen en q_R, entonces no existe ninguna asignación que satisface E_w.

E_w será la conjunción de 4 subfórmulas. Antes de especificarlas, son necesarias las siguientes aclaraciones:

- Una computación de M se representa mediante el string $\#\beta_0\#\beta_1\#\beta_2\ldots\#\beta_{p(n)}$, siendo n = |w| y β_i la configuración i-ésima de la computación. Si β_i es una configuración final, se hace $\beta_i = \beta_{i+1} = \beta_{i+2} = \ldots = \beta_{p(n)}$.
- En cada β_i, el estado corriente, el símbolo corriente y la selección no determinísitica del próximo paso (que es un número natural entre 1 y k si k es el grado de la relación de transición Δ de M), se agrupan en un solo símbolo compuesto. Así, los símbolos para representar una computación de M varían en el alfabeto Ψ formado por ternas x con: (a) un estado q de Q_M, un símbolo a de $\Gamma_M \cup \{\#\}$, y un número h de {1,…,k}, o bien (b) un «B», un símbolo a de $\Gamma_M \cup \{\#\}$, y otro «B».
- Como M trabaja en tiempo p(n), entonces no se necesitan más que p(n) símbolos para representar cada β_i. Por el mismo motivo, la representación de una computación tiene $(p(n)+1)^2$ símbolos, incluyendo los «#».
- Por cada uno de los $(p(n)+1)^2$ símbolos se va a crear una variable booleana $c_{i\,x}$, con i variando entre 0 y $(p(n)+1)^2 - 1$, $x \in \Psi$, y la variable $c_{i\,x}$ es verdadera sii el i-ésimo símbolo es el símbolo x.
- Para simplificar la escritura, $(c_{1\,x} \ldots \wedge c_{k\,x})$ y $(c_{1\,x} \vee \ldots \vee c_{k\,x})$ se abrevian $(\bigwedge_i c_{i\,x})$ y $(\bigvee_i c_{i\,x})$, respectivamente.

Se define: $E_w = E_1 \wedge E_2 \wedge E_3 \wedge E_4$, tal que:

E_1 indica que no puede ser que para una misma posición i valga tanto $c_{i\,x}$ como $c_{i\,y}$, siendo x, y, símbolos distintos de Ψ.
Más precisamente: $\bigwedge_i [(\bigvee_x c_{i\,x}) \wedge \neg (\bigvee_{x \neq y} c_{i\,x} \wedge c_{i\,y})]$, con i = 0, …, $(p(n)+1)^2 - 1$.

E_2 indica que β_0 es la configuración inicial de M con input w, y se expresa mediante otra conjunción de la forma (i) $\wedge$ (ii) $\wedge$ (iii) $\wedge$ (iv):

(i): $c_{0\,\#} \wedge c_{p(n)+1\,\#}$

(ii): $c_{1\,y1} \vee c_{1\,y2} \vee c_{1\,y3} \vee \ldots \vee c_{1\,ym}$, donde los y_i son todos los símbolos compuestos de Ψ que representan el estado inicial q_0 de M, el primer símbolo del input w, y un número posible de próximo paso de M (variando entre 1 y k).

(iii): $c_{2\,w2} \wedge c_{3\,w3} \wedge c_{4\,w4} \wedge \ldots \wedge c_{n\,wn}$

(iv): $c_{n+1\,B} \wedge c_{n+2\,B}\ \ c_{n+3\,B} \wedge \ldots \wedge c_{p(n)\,B}$

E_3 indica que la última configuración tiene un símbolo compuesto con el estado final q_A. Más precisamente:

$\vee_i (\vee_x c_{i\,x})$, con i = p(n)(p(n)+1)+1, …, $(p(n)+1)^2 - 1$, y el símbolo x de Ψ incluye el estado q_A.

Finalmente, E_4 indica que β_i le sucede adecuadamente a β_{i-1}, con i = 1, …, p(n), según la selección no determinística del próximo paso especificada en β_{i-1} y la relación de transición Δ_M de la MTN M. Más precisamente:

$\wedge_i (\vee_{v,x,y,z}\ c_{i - p(n) - 2,\, v} \wedge\ c_{i - p(n) - 1,\, x} \wedge c_{i - p(n),\, y} \wedge c_{i\,z})$, con i = p(n)+2, …, $(p(n)+1)^2 - 1$, y los símbolos v, x, y, z de Ψ son tales que se cumple el predicado f(v, x, y, z), el cual es verdadero sii z puede aparecer en la posición i de β_i estando v, x, y, en las posiciones i – 1, i, i + 1, de β_{i-1}.

Se cumple que $f(w) = E_w = E_1 \wedge E_2 \wedge E_3 \wedge E_4$, es una reducción polinomial de L a SAT:

La función f es una función total computable: existe una MTD M_f que, a partir de un input w, genera la fórmula booleana E_w, teniendo en cuenta solamente el string w y las características de la MTN M que reconoce el lenguaje L de NP en tiempo polinomial p(n).

Se cumple que $w \in L \leftrightarrow E_w \in SAT$:

- $w \in L \rightarrow$ existe una computación C de M que acepta w $\rightarrow$ por cómo se construye E_w a partir de w y M, significa que asignando según C valores de verdad adecuados a las variables $c_{i\,x}$ de E_w, se obtiene una evaluación verdadera $\rightarrow$ existe una asignación de valores de verdad que satisface $E_w \rightarrow E_w \in SAT$.
- $E_w \in SAT \rightarrow$ existe una asignación A de valores de verdad que satisface $E_w \rightarrow$ por cómo se construye E_w a partir de w y M, significa

que generando según A las distintas configuraciones β_i especificadas anteriormente, se obtiene la representación de una computación de aceptación de M a partir de w $\rightarrow$ existe una computación de M que acepta w $\rightarrow$ w ε L.

Finalmente, también se cumple que f ε P: $|E_w| = O(p(n)^2)$, donde p(n) es un polinomio. Por lo tanto M_f tarda $O(p(n)^2)$.

□

De esta manera, el lenguaje SAT está entre los lenguajes más difíciles de NP desde el punto de vista de la complejidad temporal. Asumiendo la conjetura P ≠ NP, el problema de la satisfactibilidad de las fórmulas booleanas es intratable.
Mediante las reducciones polinomiales, a partir de SAT, podrá «poblarse» el conjunto NPC, como lo fundamenta el siguiente teorema:

Teorema. Sean dos lenguajes L_1 y L_2 de NP. Si L_1 ε NPC y L_1 α_P L_2, entonces L_2 ε NPC.

En palabras, encontrando una reducción polinomial de un lenguaje NP-completo (por ejemplo SAT) a un lenguaje L de NP, se prueba que L también es NP-completo.

Prueba. Sea L algún lenguaje de NP. Como L_1 ε NPC, entonces se cumple que L α_P L_1. Como además L_1 α_P L_2, entonces por propiedad transitiva de α_P se cumple L α_P L_2. Dado que esto vale para cualquier lenguale L ε NP, entonces se cumple que L_2 es NP-completo, porque L_2 ε NP.

□

Se presentan a continuación algunos ejemplos de reducciones polinomiales para probar la pertenencia al conjunto NPC.

Ejemplo. Se cumple que CSAT = {ϑ | ϑ ε SAT y está expresada en la forma normal conjuntiva} y 3-SAT = {ϑ | ϑ ε CSAT y tiene 3 operandos en cada disyunción} son lenguajes NP-completos.
Una fórmula booleana ϑ está expresada en la forma normal conjuntiva, si ϑ es una conjunción de disyunciones de literales, siendo un literal una variable o una variable negada. Es decir,

$\vartheta = \vartheta_1 \wedge \ldots \wedge \vartheta_k$, con $\vartheta_i = (X_{i1} \vee \ldots \vee X_{ini})$, siendo $X_{ik} = x_{ik}$ ó $\neg\, x_{ik}$. Cada ϑ_i se denomina cláusula.

Se prueba que CSAT y 3-SAT están en NP (son casos particulares de SAT), y además que SAT α_P CSAT, y CSAT α_P 3-SAT (ver los ejercicios al final de esta parte del libro).

La figura 2.2.9 muestra los primeros representantes del conjunto NPC.

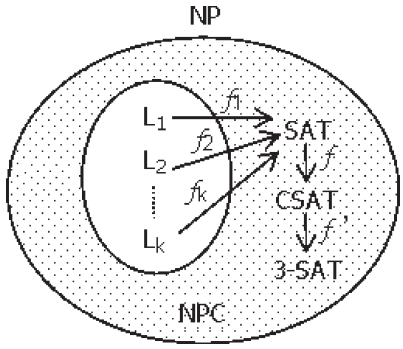


Figura 2.2.9

Las reducciones del último ejemplo consisten en *modificar componentes* de fórmulas booleanas para obtener fórmulas booleanas equivalentes con una forma determinada. La reducción del ejemplo siguiente es de naturaleza más compleja: a partir de las instancias del problema origen, se *diseñan componentes* para obtener adecuadamente las instancias del problema destino:

Ejemplo. El lenguaje VC = {(G, K) | G es un grafo que tiene un cubrimiento de vértices de tamaño K} es NP-completo.

Un grafo G = (V, E) tiene un cubrimiento de vértices V' $\subseteq$ V de tamaño K, si V' = $\{v_{i1}, v_{i2}, \ldots, v_{iK}\}$ y dado cualquier arco e = (s, t) ε E, se cumple que s ε V' o bien t ε V'. Es decir que V' es tal que |V'| = K, y tiene al menos un vértice de cada uno de los arcos de G. La figura 2.2.10 muestra ejemplos de cubrimientos de vértices de tamaño 1 y 2.

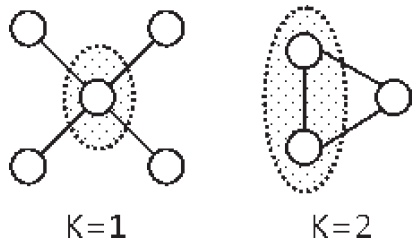


Figura 2.2.10

Se prueba fácilmente que VC ε NP. Para probar que VC ε NPH, se va a construir una reducción polinomial de 3-SAT a VC. La función de reducción propuesta es: f(ϑ) = (G, 2C). C es el número de cláusulas de ϑ, y G es un grafo que se construye del siguiente modo:

- Por cada literal de ϑ se crea un vértice de G.
- Todo par de vértices de G creados a partir de dos literales de una misma cláusula de ϑ, se unen por un arco. A este enlace se lo llamará de *tipo 1*, como así también a los triángulos creados de esta manera.
- Todo par de vértices de G creados a partir de dos literales x_i y $\neg x_i$ de ϑ, también se unen por un arco. A este enlace se lo llamará de *tipo 2*.

Por ejemplo, la figura 2.2.11 muestra el grafo que se genera a partir de la fórmula booleana
$\vartheta = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$.

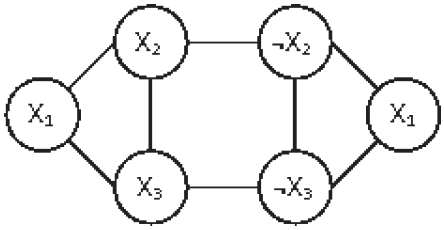


Figura 2.2.11

La función f es una función total computable:

Existe una MTD M_f que si ϑ es inválido (y por lo tanto no está en 3-SAT), entonces escribe «1» (y por lo tanto no está en VC), y que en caso contrario: (a) crea un vértice por cada literal recorriendo ϑ, (b) crea todos los enlaces de tipo 1 recorriendo una a una las ternas de vértices creadas en V, (c) crea todos los enlaces de tipo 2 recorriendo ϑ y V, y (d) escribe el número 2C contabilizando las cláusulas de ϑ.

Se cumple que $\vartheta \in$ 3-SAT $\leftrightarrow$ (G, 2C) $\in$ VC:

$\vartheta \in$ 3-SAT:
$\rightarrow \vartheta = \vartheta_1 \wedge \vartheta_2 \wedge \vartheta_3 \wedge \ldots \vartheta_C$, siendo $\vartheta_i = (X_{i1} \vee X_{i2} \vee X_{i3})$ y $X_{ih} = x_{ih}$ ó $\neg x_{ih}$ y existe una asignación de valores de verdad que satisface ϑ (por la forma de ϑ, en toda cláusula ϑ_i al menos un literal X_{ih} debe recibir el valor verdadero).
$\rightarrow$ El siguiente subconjunto de vértices V' de V, es un cubrimiento de G de tamaño 2C: todo vértice v asociado a un literal que recibe el valor falso se incluye en V', y si luego de esta inclusión hay triángulos de tipo 1 que no tienen exactamente dos vértices en V' (caso de cláusulas con dos o tres literales con valor verdadero), entonces se agregan vértices cualesquiera de dichos triángulos hasta lograrlo. Así se cumple que |V'| = 2C. Y también se cumple que V' cubre a G: los enlaces de tipo 1 están cubiertos porque V' tiene dos vértices de cada triángulo de tipo 1, y los enlaces de tipo 2 están cubiertos porque si un literal x_{ih} recibe el valor verdadero, entonces el literal $\neg x_{ih}$ recibe el valor falso, y por lo tanto, por lo indicado antes, el vértice asociado a $\neg x_{ih}$ pertenece a V'.
$\rightarrow$ (G, 2C) $\in$ VC.

(G, 2C) ε VC:
$\rightarrow$ G tiene un cubrimiento de vértices V' de tamaño 2C.
$\rightarrow$ La siguiente asignación A de valores de verdad satisface $\vartheta = \vartheta_1 \wedge \vartheta_2 \wedge \ldots \wedge \vartheta_C$, con $\vartheta_i = (X_{i\,1} \vee X_{i\,2} \vee X_{i\,3})$ y $X_{i\,h}$ es $x_{i\,h}$ ó $\neg\, x_{i\,h}$: a los literales asociados a los vértices que no están en V' les asigna el valor verdadero, y al resto de los literales les asigna consistentemente valores cualesquiera verdadero o falso. Se cumple que A satisface ϑ porque: (a) Como V' tiene exactamente dos vértices por triángulo de tipo 1, entonces A asigna verdadero al menos a un literal de cada cláusula. (b) A es consistente, no puede darse el caso de dos cláusulas en las que $x_{i\,h}$ sea verdadera en una y $\neg\, x_{i\,h}$ sea verdadera en la otra, porque si no el enlace de tipo 2 asociado al par de literales $(x_{i\,h}, \neg\, x_{i\,h})$ no estaría cubierto por V' (absurdo).
$\rightarrow \vartheta \varepsilon$ 3-SAT.

Finalmente, se cumple que f ε P:

La MTD M_f referida antes trabaja en tiempo $O(n^2)$. Por un lado, chequear si ϑ es inválido tarda O(n), y escribir «1» tarda O(1). Por otro lado, crear un vértice por literal tarda O(n), crear todos los enlaces de tipo 1 tarda O(n), crear todos los enlaces de tipo 2 tarda $O(n^2)$, y escribir el número 2C tarda O(n).

De esta manera se probó que el problema de decidir si un grafo tiene un cubrimiento de vértices de tamaño K es intratable, asumiendo que P $\neq$ NP.

Un tercer tipo de reducción, más simple que las anteriores, consiste directamente en reducir un problema a otro similar. La reducción polinomial del lenguaje HC al lenguaje TSP presentada previamente, es un ejemplo de este tipo de reducción. Se muestra a continuación otro ejemplo, que prueba que el problema de decidir si un grafo tiene un clique de tamaño K es intratable, asumiendo que P $\neq$ NP.

Ejemplo. Sea otra vez el lenguaje CLIQUE = {(G, K) | G tiene un clique de tamaño K}. Ya se probó que CLIQUE ε NP. Se va a probar que VC α_P CLIQUE, y como VC ε NPC, entonces se demostrará que CLIQUE también es un lenguaje NP-completo.

La función de reducción f propuesta es: f((G, K)) = (G^C, m – K), siendo G^C el grafo «complemento» de G, es decir que tiene los mismos vértices que G y sólo los arcos que no tiene G. El número m es, como siempre, la cantidad de vértices del grafo G (ver la figura 2.2.12).

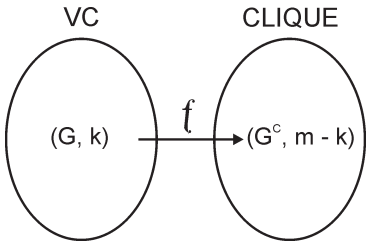


Figura 2.2.12

Queda como ejercicio para el lector, probar que f es una función total computable y que pertenece a P.

La figura 2.2.13 muestra dos casos de aplicación de la función de reducción f sobre pares (G, K). En el primer caso, de un grafo de 4 vértices con un cubrimiento de 2 vértices, se pasa a un grafo «complemento» con un clique de tamaño 2. En el segundo caso, de un grafo de 5 vértices con un cubrimiento de 2 vértices, se pasa a un grafo «complemento» con un clique de tamaño 3.

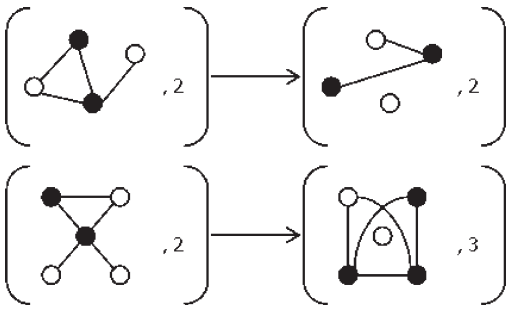


Figura 2.2.13

Formalmente, se cumple que (G, K) ε VC $\leftrightarrow$ (G^C, m – K) ε CLIQUE:

(G, K) ε VC:
$\rightarrow$ G tiene un cubrimiento de vértices V' de tamaño K.
$\rightarrow$ (V – V') es un clique de G^C de tamaño (m – K): (a) (V – V') tiene (m – K) vértices. (b) Supóngase que G^C no tiene el arco (v_i, v_h), siendo v_i y v_h vértices de (V – V'). Entonces (v_i, v_h) es un arco de G, siendo v_i y v_h vértices que no están en V', por lo que V' no es un cubrimiento de G (absurdo).
$\rightarrow$ (G^C, m – K) ε CLIQUE.

(G^C, m – K) ε CLIQUE:
$\rightarrow$ G^C tiene un clique V'' de tamaño (m – K).
$\rightarrow$ (V – V'') es un cubrimiento de vértices de G de tamaño K: (a) (V – V'') tiene (m – (m – K)) = K vértices. (b) Supóngase que G tiene el arco (v_i, v_h), siendo v_i y v_h vértices que no están en (V – V''). Entonces (v_i, v_h) no es un arco de G^C, siendo v_i y v_h vértices que están en V'', por lo que V'' no es un clique de G^C (absurdo).
$\rightarrow$ (G, K) ε VC .

Considerando los distintos grados de dificultad de los tipos de reducciones polinomiales que se mencionaron previamente, entonces una estrategia posible para probar que un lenguaje L de NP es NP-completo sería intentar encontrar una reducción polinomial de otro lenguaje NP-completo L' a L, primero recurriendo a un lenguaje L' que represente un problema similar al representado por L, luego recurriendo a la técnica de modificación de componentes sobre los elementos de L', y finalmente recurriendo a la técnica de diseño de componentes a partir de los elementos de L'.
Cabe remarcar que para todo par de lenguajes NP-completos conocidos L_1 y L_2, se cumple que existe una biyección f entre ellos, cumpliéndose que L_1 se reduce polinomialmente a L_2 por medio de f y que L_2 se reduce polinomialmente a L_1 por medio de f^{-1}. Se define en este caso que los lenguajes L_1 y L_2 son *p-isomorfos*.
Es decir, no se conocen pares de lenguajes NP-completos que no sean p-isomorfos. Existe una conjetura, la Conjetura de Hartmanis y Berman, que formula justamente que todos los lenguajes NP-completos son dos a dos p-isomorfos. Si se comprobara esta conjetura, se resolvería el problema abierto de la relación entre P y NP, como se plantea en el siguiente teorema:

Teorema. Si todos los lenguajes NP-completos son dos a dos p-isomorfos, entonces P ≠ NP.

Prueba. Si P = NP, entonces P = NPC (se cumple que P ⊆ NPC porque, como se probó en una sección anterior, todos los lenguajes de P son reducibles unos a otros en tiempo polinomial). Por lo tanto, los lenguajes finitos son NP-completos. Pero como un lenguaje finito y un lenguaje infinito no pueden ser p-isomorfos, entonces no todos los lenguajes NP-completos son dos a dos p-isomorfos.

□

Otra caracterización de la clase NPC se relaciona con la densidad de los lenguajes que la componen: todos los lenguajes NP-completos que se conocen son *densos*, es decir, para todo n, la cantidad de sus strings de longitud a lo sumo n no se puede acotar por un polinomio p(n). Lo interesante de esta observación es que se prueba que si existe un lenguaje NP-completo no denso (se denomina *esparso*), entonces P = NP.

2.2.2.3. Problemas NP-intermedios

Asumiendo P ≠ NP, se prueba que además de P y NPC, la clase NP incluye un tercer conjunto de problemas, NPI, que serían los problemas de dificultad «intermedia». La figura 2.2.14 ilustra la estructura interna de NP con este nuevo conjunto de problemas.

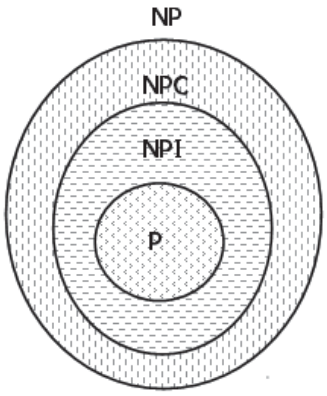


Figura 2.2.14

La existencia de NPI se puede demostrar utilizando un teorema que establece que si B es un lenguaje recursivo fuera de P, entonces existen dos lenguajes D ε P y A $\notin$ P, tales que A = D $\cap$ B, y además A se reduce polinomialmente a B pero B no se reduce polinomialmente a A. De este modo, asumiendo P $\neq$ NP:

- Sea B ε NPC (cumple la hipótesis).
- Entonces por el teorema, A ε NP (A α_P B y B ε NP).
- Entonces A $\notin$ NPC (no existe una reducción polinomial de B a A, y B ε NP).
- Finalmente, como A $\notin$ P, se cumple que A ε NP – (P $\cup$ NPC) = NPI.

Existen problemas en NP a los que no se les han encontrado soluciones determinísticas polinomiales, ni pruebas de pertenencia a la clase NPC. Así, son candidatos a pertenecer a la clase NPI.
Un ejemplo clásico de posible problema de NPI es el problema de determinar si dos grafos son isomorfos. Dos grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ son isomorfos, si $|V_1| = |V_2|$ y existe una permutación Π de los números 1, 2, 3, …, m, tal que $\forall$ i, k: (i, k) ε $E_1 \leftrightarrow (\Pi(i), \Pi(k)) \varepsilon E_2$. Es decir, los grafos G_1 y G_2 son isomorfos si son iguales a no ser por la identificación de sus arcos. El lenguaje que representa el problema es $L = \{(G_1, G_2) \mid G_1$ y G_2 son grafos isomorfos$\}$.
La solución determinística natural al problema de los grafos isomorfos tarda tiempo exponencial. Al final de esta parte del libro se pide probar que el problema está en NP. Por otro lado, no se ha encontrado ninguna reducción polinomial de un problema NP-completo a dicho problema.
Como en el caso de la clase NPC, NPI también se puede caracterizar por las densidades de sus lenguajes. Se prueba que si P $\neq$ NP, entonces existe un lenguaje esparso en NPI sii existe un lenguaje *tally* en NPI (los lenguajes tally son casos particulares de lenguajes esparsos, cuyos strings se generan a partir de un alfabeto de un solo símbolo). Como se verá en la próxima sección, esta relación hace plausible la existencia de lenguajes esparsos en NPI, y así refuerza la idea de separación entre NPI y NPC.

2.2.3. El exterior de la clase NP

Para cerrar este capítulo, en las siguientes secciones se describen aspectos generales de lo que existe más allá de la clase NP. Primero se caracteriza el conjunto de lenguajes tales que sus complementos están

en NP. Luego se hace una referencia suscinta a los representantes más difíciles de EXP.

2.2.3.1. La clase CO-NP

Formalmente no puede establecerse relación alguna entre NP y CO-NP, siendo CO-NP el conjunto de los lenguajes recursivos tales que sus complementos pertenecen a NP.

Se cumple que $P \subseteq (NP \cap CO\text{-}NP)$. $P \subseteq NP$ por definición, y también vale $P \subseteq CO\text{-}NP$ porque el conjunto P es cerrado con respecto al complemento: $L \varepsilon P \rightarrow L^C \varepsilon P \rightarrow L^C \varepsilon NP \rightarrow L \varepsilon CO\text{-}NP$.

También se prueba que la asunción $NP \neq CO\text{-}NP$ es más fuerte que la asunción P NP:

Teorema. $NP \neq CO\text{-}NP \rightarrow P \neq NP$.

Prueba. Si P = NP, como P es cerrada con respecto al complemento, entonces también lo es la clase NP, es decir, NP = CO-NP.

□

La figura 2.2.15 ilustra con más detalle el mapa de la complejidad temporal presentado anteriormente, asumiendo $NP \neq CO\text{-}NP$.

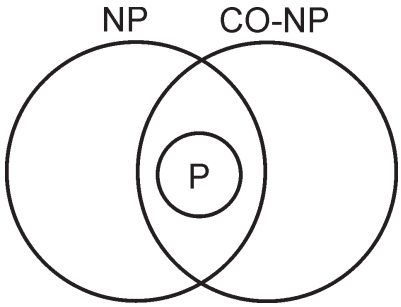


Figura 2.2.15

Considerando este mapa, NPC y (NP $\cap$ CO-NP) son disjuntos. Intuitivamente, se tiene más información de los problemas en (NP $\cap$ CO-NP) que de los problemas en NP – (NP $\cap$ CO-NP), y al ser NPC la clase de los problemas más difíciles de NP, entonces los problemas de (NP $\cap$ CO-NP) deberían ser de dificultad baja (los de P) o intermedia (los de NPI). Formalmente:

Teorema. Si L ε (NPC $\cap$ CO-NP), entonces NP = CO-NP.

Prueba. Se va a probar solamente que NP $\subseteq$ CO-NP. La inclusión inversa se prueba de un modo similar y queda como ejercicio para el lector:

Sea L' ε NP. Como L es NP-completo, entonces L' α_P L, o lo que es lo mismo, L' ^{C} α_P L^C. Dado que L^C ε NP, entonces por la reducción se cumple que L' ^{C} ε NP, y por lo tanto L' ε CO-NP.

□

Así, una manera de «probar» que un lenguaje L no es NP-completo, asumiendo NP ≠ CO-NP, es probar que L y L^C están en NP. La figura 2.2.16 muestra la nueva versión del mapa de la complejidad temporal, asumiendo la conjetura NP ≠ CO-NP.

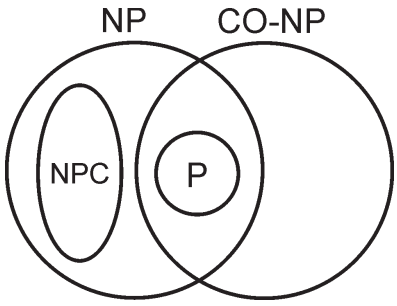


Figura 2.2.16

Existe una jerarquía temporal que caracteriza específicamente la relación entre NP y CO-NP. Se denomina Jerarquía Booleana, y se denota con BH. Las primeras clases de esta jerarquía son:

- $BH_1 = NP$.
- $BH_2 = \{L_1 \cap L_2^C \mid L_1 \varepsilon BH_1 \text{ y } L_2 \varepsilon NP\}$.
- $BH_3 = \{L_1 \cup L_2 \mid L_1 \varepsilon BH_2 \text{ y } L_2 \varepsilon NP\}$.
- $BH_4 = \{L_1 \cap L_2^C \mid L_1 \varepsilon BH_3 \text{ y } L_2 \varepsilon NP\}$.
- **.....**

BH es la unión infinita de las clases BH_i, para $i \geq 1$. Se prueba que BH es la menor clase de problemas que incluye a NP y es cerrada con respecto a las operaciones de unión, intersección y complemento (de esta propiedad proviene el nombre de la jerarquía, dado que BH es la clausura booleana de NP). En el siguiente ejemplo se presenta un problema conocido de la clase BH_2:

Ejemplo. Sea el siguiente problema: «Dadas las fórmulas booleanas ϑ_1 y ϑ_2 en la forma normal conjuntiva, ¿acaso ϑ_1 es satisfactible y ϑ_2 no lo es?». El lenguaje que representa el problema es SAT-2 = $\{(\vartheta_1, \vartheta_2) \mid \vartheta_1$ y ϑ_2 son fórmulas booleanas en la forma normal conjuntiva, ϑ_1 es satisfactible y ϑ_2 no es satisfactible$\}$. Se prueba que SAT-2 ε BH_2:

Sea L_1 el lenguaje de los pares $(\vartheta_1, \vartheta_2)$ de fórmulas booleanas en forma normal conjuntiva, tales que ϑ_1 es satisfactible.

Y sea L_2 el lenguaje de los pares $(\vartheta_1, \vartheta_2)$ de fórmulas booleanas en forma normal conjuntiva, tales que ϑ_2 es satisfactible.

Por lo tanto SAT-2 = $L_1 \cap L_2^C$, y como los lenguajes L_1 y L_2 están en NP, entonces SAT-2 ε BH_2.

Es más, todos los lenguajes de BH_2 se reducen polinomialmente a SAT-2 (SAT-2 es BH_2 - completo). Los problemas C-completos, cualquiera sea la clase C, se tratan más adelante.

2.2.3.2. Los problemas más difíciles de la clase EXP

En EXP se distinguen dos conjuntos de lenguajes:

- $LEXP = \bigcup_{k>0} DTIME(2^x)$, con $x = k.n$
- $PEXP = \bigcup_{k>0} DTIME(2^y)$, con $y = n^k$

Para el caso no determinístico, los conjuntos NLEXP y NPEXP se definen de manera similar.
Se prueba que $P \subset LEXP \subset PEXP$, y que $NP \subset NLEXP \subset NPEXP$. Por otro lado, no puede demostrarse si NLEXP incluye estrictamente a LEXP. En cambio, sí se puede probar que:

- Si $LEXP \neq NLEXP$, entonces $P \neq NP$.
- $LEXP \neq NLEXP$ sii (NP – P) incluye un lenguaje esparso.
- $LEXP \neq NLEXP$ sii (NP – P) incluye un lenguaje tally.

Dados estos enunciados, y lo formulado previamente de que $P \neq NP$ implica que existe un lenguaje esparso en NPI sii existe un lenguaje tally en NPI, entonces sería plausible la existencia de lenguajes esparsos dentro de la clase NPI.

Se define una Jerarquía Temporal Exponencial de la siguiente manera. Dados $y = n^k$, $z_1 = 2^y$, $z_2 = 2^{z1}$, $z_3 = 2^{z2}$, $z_4 = 2^{z3}$, …, las clases de esta jerarquía son:

- $PEXP = \bigcup_{k>0} DTIME(z_1)$
- $2\text{-}PEXP = \bigcup_{k>0} DTIME(z_2)$
- $3\text{-}PEXP = \bigcup_{k>0} DTIME(z_3)$
- $4\text{-}PEXP = \bigcup_{k>0} DTIME(z_4)$
- **……..**

De modo similar se puede definir la jerarquía NPEXP, 2-NPEXP, 3-NPEXP, 4-NPEXP, …

La unión de todas estas clases de lenguajes se conoce como el conjunto de los lenguajes *elementales*.

Terminando esta sección, se mencionan a continuación dos problemas de EXP, que se prueba que no tienen solución eficiente:

Ejemplo. Dos problemas clásicos de la literatura, uno de la teoría de los lenguajes formales, y el otro de la lógica, se prueba que tienen solución que tarda mínimamente tiempo exponencial.

El problema de las expresiones regulares con exponenciación, consiste en decidir si una expresión regular con exponenciación denota todos los strings sobre un alfabeto.
Las expresiones regulares constituyen una forma alternativa para representar los lenguajes regulares o de tipo 3. Se construyen a partir de los símbolos de los alfabetos sobre los cuales se definen los lenguajes, y determinados operadores. En particular, el operador de exponenciación **«↑»** se utiliza de la siguiente manera: «a ↑ i» se interpreta como i veces seguidas el símbolo «a».
El lenguaje asociado al problema se denomina REX. Se prueba que REX se resuelve como mínimo en tiempo determinístico exponencial. Más precisamente, se prueba que toda solución determinística de REX requiere espacio exponencial (como se verá en los capítulos sobre complejidad espacial, espacio exponencial implica tiempo exponencial): se demuestra que todo lenguaje decidible determinísticamente en espacio exponencial, se puede reducir en tiempo polinomial al lenguaje REX.

Por su parte, *el problema de decisión para los números reales con adición*, consiste en determinar la verdad o falsedad de una fórmula de la teoría de los números reales con adición.
La teoría de los números reales con adición es decidible, y se representa de la siguiente manera:

$$(R, +, =, <, 0, 1),$$

siendo R el dominio de los números reales, y «+», «=», «<», «0» y «1» la operación, los predicados y las constantes definidas en la teoría, respectivamente.
Se prueba que el problema de decisión para los números reales con adición se resuelve como mínimo en tiempo no determinístico exponencial: se demuestra que se puede reducir en tiempo polinomial al lenguaje que representa dicho problema, cualquier lenguaje decidible en tiempo no determinístico exponencial.

CAPÍTULO 2.3. INTRODUCCIÓN A LA COMPLEJIDAD ESPACIAL

De la misma forma que en el Capítulo 2.1, en este capítulo se presentan al principio las definiciones fundamentales acerca de la complejidad computacional espacial (Sección 2.3.1). Se precisa cómo se mide el espacio que consume una máquina de Turing, y cómo se caracteriza una clase de problemas desde el punto de vista de la complejidad espacial. Además, se introduce como modelo estándar, las máquinas de Turing con una cinta de input de solo lectura (máquinas de Turing *off-line*), para permitir trabajar con espacios logarítmicos. Esta elección se mantiene consistente con el modelo de ejecución considerado en la complejidad temporal.
En la Sección 2.3.2 se introduce la jerarquía de las clases espaciales de problemas. Se describen sus características principales, y se mencionan las clases más importantes: las de los problemas que se resuelven en espacio determinístico y no determinístico logarítmico (DLOGSPACE y NLOGSPACE, respectivamente), y las de los problemas que se resuelven en espacio determinístico y no determinístico polinomial (PSPACE y NPSPACE, respectivamente).
Como el espacio acotado de una máquina de Turing no asegura que se detenga, se demuestra que de todos modos siempre existe una máquina de Turing equivalente que consume el mismo espacio y siempre se detiene, lo que permite enmarcar el estudio de la complejidad espacial dentro de la clase de los problemas decidibles. La prueba se basa en una técnica que se utiliza a lo largo de éste y el siguiente capítulo, que consiste en acotar el número máximo de configuraciones posibles de una máquina de Turing según el espacio en que trabaja.
Se destaca que a diferencia de lo que sucede con el tiempo, en que no se conoce la relación entre P y NP, en este caso se cumple que las clases PSPACE y NPSPACE son iguales. También se prueba que el espacio no determinístico es cerrado con respecto al complemento. Ambas consideraciones revelan que la potencia computacional del no determinismo es mayor en el ámbito del tiempo que en el del espacio.

2.3.1. Definiciones iniciales

Para el estudio de la complejidad espacial de los problemas se utilizan las MT *off-line*, que son MT con una *cinta de input de solo lectura*, en la

que ningún símbolo puede ser modificado, y sobre la que el cabezal sólo se mueve a lo largo del input (más el símbolo «B» de la izquierda y el símbolo «B» de la derecha).
Se verá enseguida la utilidad de la cinta de input de solo lectura. El resto de las cintas se denominan *cintas de trabajo* o *cintas auxiliares*. Cuando es necesario, se emplea también una cinta de output.
Mientras que el tiempo de trabajo de una MT se refiere a los pasos que ejecuta, el espacio se define en términos de las celdas que recorre. Formalmente:

Definición. Sea la función $S: N \rightarrow N^+$. Una MT M trabaja en espacio S(n), sii tiene una cinta de input de solo lectura, y para todo input w, tal que $|w| = n$, M utiliza a lo sumo S(n) celdas en todas las cintas de trabajo (en su única computación si M es una MT determinística, o en cada una de sus computaciones si M es una MT no determinística).

□

De modo similar se define una MT que trabaja en espacio O(S(n)). El codominio de la función S es N^+ porque toda MT utiliza al menos una celda en toda cinta de trabajo.
Los lenguajes reconocibles por MT que trabajan en espacio O(S(n)), pertenecen a una misma clase o conjunto de lenguajes:

Definición. Un lenguaje (o problema de decisión) pertenece a la clase DSPACE(S(n)), sii existe una MTD con una cinta de input de solo lectura y K cintas de trabajo, que lo reconoce (o resuelve) en espacio O(S(n)). La misma definición vale para la clase NSPACE(S(n)), considerando las MTN.

□

Notar que en las definiciones anteriores no se explicita que las MT se detienen a partir de todos los inputs. Se probará al final de este capítulo que si existe una MT que trabaja en espacio S(n), entonces existe una MT equivalente que trabaja en espacio S(n) y se detiene a partir de todos los inputs, por lo que las clases de complejidad espacial están incluidas, como corresponde, en el conjunto R de los lenguajes recursivos.
Como en la complejidad temporal, las MT estándar empleadas para el estudio de la complejidad espacial son las MTD con K cintas. Según la simulación desarrollada en la primera parte del libro, el espacio consumido es el mismo utilizando MT con cualquier cantidad finita de

cintas. Para simplificar la presentación, se trabajará en general con MT con una sola cinta de trabajo.
El uso de una cinta de input de solo lectura permite trabajar con órdenes espaciales menores que el lineal, como se aprecia en el siguiente ejemplo:

Ejemplo. Se prueba que el lenguaje L = {wcw^R | w ε {a, b}*}, siendo w^R el string inverso de w, pertenece a la clase DSPACE(n). Sea la siguiente MTD M:

1) Copiar el comienzo del input x hasta antes del símbolo «c» en la cinta de trabajo 1, dejando el cabezal de esta cinta apuntando al último símbolo copiado. Rechazar si se encuentran símbolos distintos de «a», «b» y «c», o si no se detecta exactamente un símbolo «c». Posicionar el cabezal de la cinta de input a la derecha del símbolo «c».
2) Comparar símbolo a símbolo el contenido de la cinta de input, yendo a la derecha, con el contenido de la cinta de trabajo 1, yendo a la izquierda, hasta encontrar un «B» en cada cinta. Rechazar si se detecta a lo largo de la comparación un par de símbolos distintos, y aceptar en caso contrario.

Se comprueba fácilmente que L(M) = L y que M trabaja en espacio determinístico O(n). De todos modos, existe una MTD M' que reconoce el lenguaje L en menos espacio que M. A partir de un input x, la MT M' hace:

1) Escribir la cantidad i de símbolos de la parte izquierda de x, sin incluir el símbolo «c», en la cinta de trabajo 1, y la cantidad k de símbolos de la parte derecha de x, sin incluir el símbolo «c», en la cinta de trabajo 2. Rechazar si se encuentran símbolos distintos de «a», «b» y «c», o si no hay exactamente un símbolo «c». Aceptar si i = k = 0, y rechazar si i ≠ k.
2) Dado el input x, comparar su símbolo i de la parte izquierda (i según la cinta de trabajo 1) con su símbolo (k – i + 1) de la parte de derecha (k según la cinta de trabajo 2), y si son distintos rechazar. Si no, hacer i := i – 1 en la cinta de trabajo 1. Si i = 0 aceptar, y en caso contrario volver a (2). (Por ejemplo, si |x| = 11, las partes izquierda y derecha tienen 5 símbolos cada una, y así primero se compara el símbolo 5 de la parte izquierda con el símbolo 1 de la parte derecha,

después el símbolo 4 de la parte izquierda con el símbolo 2 de la parte derecha, después el 3 con el 3, el 2 con el 4, y finalmente el 1 con el 5.)

Se cumple que L(M') = L. Además, M' trabaja en espacio O(log(n)), por lo que el lenguaje L pertenece a la clase DSPACE(log(n)):

- Los números, como siempre, se representan en notación no unaria.
- Calcular y escribir los números i y k no consume más que O(log(|x|)) celdas. Lo mismo ocurre con las operaciones de (2).
- Finalmente, la comparación de símbolos en (2) requiere tener en memoria sólo dos símbolos, lo que consume espacio constante.

Cuando se cumple S(n) > n, se puede utilizar una MT con una cinta de input común. La clase DSPACE(log(n)) se conoce también como DLOGSPACE. Se verá después que el espacio logarítmico desempeña un papel muy importante en la complejidad espacial.

La técnica utilizada en el ejemplo es típica de la complejidad espacial. Se utilizan punteros sobre la cinta de input de solo lectura, se trabaja con contadores representados en bases adecuadas, se almacena en las cintas de trabajo sólo pequeños substrings del input, etc. El siguiente es otro ejemplo en que se observan estas características:

Ejemplo. Sea el lenguaje CON = {(G, s, t) | G es un grafo orientado, *s* y *t* son el primero y último vértice de G, y existe un camino en G desde s hasta t}. El lenguaje CON representa el problema de la alcanzabilidad.

Se prueba que CON ε NSPACE(log(n)). Sea la siguiente MTN M, con un contador C en la cinta de trabajo 3 (al comienzo C = 1, y como siempre, la cantidad de vértices del grafo G es m):

1) Escribir el vértice s de G como v_1 en la cinta de trabajo 1.
2) Escribir no determinísticamente un vértice v_2 de G en la cinta de trabajo 2.
3) Si (v_1, v_2) no es un arco de G, rechazar. Si v_2 = t, aceptar. Hacer C := C + 1, y si C = m, rechazar.
4) Reemplazar v_1 por v_2 en la cinta de trabajo 1, y volver a (2).

Se cumple que CON = L(M):

Sólo si existe un camino de *s* a *t* en el grafo G, el paso (2) de M lo irá recorriendo vértice a vértice. En el peor caso, la MT M hace (m – 1) iteraciones hasta aceptar o rechazar en (3).

Además, M trabaja en espacio no determinístico O(log(n)):

M no construye el camino buscado (esto consumiría espacio lineal), sino que le alcanza con tener en memoria sólo un par de vértices, cada uno de los cuales mide O(log(n)) celdas. El contador C también ocupa espacio O(log(n)).

La clase NSPACE(log(n)) se conoce también como NLOGSPACE. Se prueba fácilmente que, además de pertenecer a NLOGSPACE, el lenguaje CON también pertenece a P, lo que va a contribuir a determinar la relación entre las clases de complejidad NLOGSPACE y P.

2.3.2. Generalidades de la jerarquía espacial

Además de DLOGSPACE y NLOGSPACE, otro par de clases de complejidad espacial que se distinguen dentro de R son PSPACE y NPSPACE.
PSPACE es la clase de los problemas que se resuelven en espacio determinístico polinomial, y NPSPACE es la clase de los problemas que se resuelven en espacio no determinístico polinomial. Formalmente:

Definición. PSPACE = $\bigcup_{i \geq 0}$ DSPACE(n^i), y NPSPACE = $\bigcup_{i \geq 0}$ NSPACE(n^i).

□

A diferencia de lo que sucede con el tiempo, en que no se sabe si vale la inclusión estricta P $\subset$ NP, en este caso se cumple que PSPACE = NPSPACE.
Por otro lado, siendo CO-NSPACE(S(n)) el conjunto de los lenguajes tales que sus complementos están en NSPACE(S(n)), se cumple que NSPACE(S(n)) = CO-NSPACE(S(n)), fijando algunas restricciones

básicas sobre la función S(n). Recordar que, en el marco de la complejidad temporal, ni siquiera se conoce la relación entre NP y CO-NP.
De los últimos dos enunciados, que luego se probarán, se desprende que la potencia computacional del no determinismo es mayor en el ámbito del tiempo que en el del espacio.
En lo que sigue, se presentan algunas características generales de la jerarquía espacial. Si bien sólo se trata el espacio determinístico, todo aplica también al espacio no determinístico.
En primer lugar, cabe destacar lo mismo que se mencionó en la introducción a la complejidad temporal: que valga $S_1(n) = O(S_2(n))$, dadas dos funciones S_1 y S_2, no significa que la clase $DSPACE(S_2(n))$ incluya estrictamente a la clase $DSPACE(S_1(n))$. Por ejemplo:

Teorema de *Linear Tape Compression* (compresión lineal). Si existe una MTD M_1 que trabaja en espacio $cS(n)$, con $c > 0$, entonces existe una MTD M_2 equivalente que trabaja en espacio $S(n)$.

Prueba. La idea general es la misma que la del Teorema de *Linear Speed Up* visto para el tiempo. Se construye una MTD M_2 equivalente a M_1 tal que sus alfabetos Σ_2 y Γ_2 tienen como símbolos secuencias de k símbolos de los alfabetos Σ_1 y Γ_1, para un k determinado que depende de c. Las cintas de M_2 tienen k tracks. Se logra que lo que en M_1 ocupa c celdas, en M_2 ocupe sólo una.

□

El «salto» de una clase espacial a otra se produce por medio de una función mayor que un factor constante. Se prueba que si $\inf_{n \to \infty}(S_1(n) / S_2(n)) = 0$, $S_1(n)$ y $S_2(n)$ no son menores que $\log_2 n$, y $S_2(n)$ «se comporta bien», entonces se cumple que $DSPACE(S_1(n)) \neq DSPACE(S_2(n))$. En particular, si $S_1(n) = O(S_2(n))$, entonces $DSPACE(S_1(n)) \subset DSPACE(S_2(n))$.

Notar que la manera de «saltar» a una clase espacial mayor es menor que lo requerido en la jerarquía temporal. Debe darse directamente que $S_2(n)$ sea significativamente mayor que $S_1(n)$ cuando n tiende a infinito. Que $S_2(n)$ se «comporte bien» significa que debe ser una función *espacio-construible*:

Definición. Una función S: N $\rightarrow$ N^+ es espacio-construible, si existe una MTD M que para todo input w tal que |w| = n, M trabaja en espacio exactamente S(n) y se detiene.

□

Las funciones espacio-construibles S(n) permitirán marcar S(n) celdas, lo que resultará muy útil en las pruebas de los teoremas. Y al igual que sucede en la complejidad temporal cuando no se utilizan funciones tiempo-construibles, trabajar con funciones que no sean espacio-construibles puede provocar comportamientos extraños en la jerarquía espacial, tales como *gaps*.

Ejercicio. *Probar que si S(n) es una función espacio-construible, entonces existe una MTD que imprime en una de sus cintas auxiliares exactamente S(n) marcas «X».*

□

Todas las funciones con las que se trabaja habitualmente en la complejidad espacial, como log(n), n^k, 2^n, n!, etc., son espacio-construibles. Además, no es común tratar con funciones menores que log(n). Por lo tanto, la hipótesis de que S(n) sea una función espacio-construible, mayor o igual que log(n), que se establecerá en general en los teoremas, no será restrictiva.
Como ejemplo de aplicación de los enunciados anteriores definidos en el marco de la jerarquía espacial, se puede establecer que DLOGSPACE está incluido estrictamente en PSPACE:

Ejemplo. Se cumple que DLOGSPACE $\subset$ PSPACE. En realidad, DLOGSPACE $\subset$ DSPACE(n^k), para cualquier k natural mayor que cero: (a) log(n) y n^k son funciones no menores que log(n). (b) n^k es una función espacio-construible. (c) Se cumple que $\inf_{n \rightarrow \infty} (\log(n) / n^k) = 0$. (d) Se cumple que $\log(n) = O(n^k)$.

La jerarquía espacial es «densa». El siguiente teorema refuerza esta idea:

Teorema. Si S: N $\rightarrow$ N^+ es una función total computable, entonces existe un lenguaje recursivo L tal que L $\notin$ DSPACE(S(n)).

Prueba. Es muy similar a la del teorema análogo visto para el tiempo. Queda como ejercicio para el lector.

□

Se prueba a partir de un teorema denominado Teorema de la Unión, que existe una función total computable S(n) tal que DSPACE(S(n)) = PSPACE. Por lo tanto, como corolario del teorema anterior, PSPACE $\subset$ R, y dado que PSPACE = NPSPACE (esto se prueba más adelante), también se cumple NPSPACE $\subset$ R.
En tiempo T(n) no pueden recorrerse más de T(n) celdas, y por lo tanto P $\subseteq$ PSPACE, y de una manera similar se infiere que NP $\subseteq$ PSPACE (no se sabe si las inclusiones son estrictas). Por lo tanto, a partir del corolario anterior se puede establecer que hay lenguajes recursivos por fuera de P y NP.
Será lo mismo hablar de R que de la clase espacial exponencial EXPSPACE, la cual nuclea a todos los lenguajes recursivos reconocibles en espacio exponencial.
Para cerrar este capítulo introductorio, se prueba a continuación lo que se anticipó en las definiciones iniciales, que la jerarquía espacial se enmarca dentro del conjunto de los lenguajes recursivos.

Teorema. Si un lenguaje L es aceptado por una MT M que trabaja en espacio S(n) > $\log_2 n$, siendo S(n) una función espacio-construible, entonces L es aceptado por una MT M' que trabaja en espacio S(n) y se detiene a partir de todos los inputs.
Las restricciones sobre S(n) no quitan generalidad al teorema, dado que todas las funciones con las que se trabaja habitualmente las satisfacen. Además, tener en cuenta que pueden producirse anomalías en la jerarquía espacial cuando se emplean funciones que no son espacio-construibles. De todos modos, el teorema también se puede probar sin la premisa de que S(n) sea espacio-construible, pero la demostración se torna más compleja.

Prueba. Sea M una MT *off-line* que reconoce el lenguaje L, con s estados y t símbolos (es decir que |Q| = s y |Γ| = t), y tal que trabaja en espacio S(n) > $\log_2 n$, siendo S(n) espacio-construible.

Se cumple que a partir de un input w, el número máximo de configuraciones distintas por las que puede pasar la MT M es N = (n+2).S(n).s.$t^{S(n)}$, dado que el cabezal a la cinta de input se mueve entre

(n+2) celdas y el cabezal a la cinta de trabajo entre S(n) celdas. Si M hace más de N pasos significa que «entra en loop».

Para simplificar ésta y las próximas demostraciones, se utilizará de ahora en más una cota mayor que N, el valor $(4st)^{S(n)}$. Siendo $S(n) > \log_2(n)$, se prueba que:

$$(4st)^{S(n)} > (n+2).S(n).s.t^{S(n)}.$$

La siguiente MT M' acepta L en espacio S(n) y se detiene a partir de todos los inputs. El input w está en la cinta de input de solo lectura. Se simula M en la cinta de trabajo 1. Hay un contador C en la cinta de trabajo 2 escrito en base 4st, con valor inicial cero, que lleva la cuenta de los pasos simulados de M:

1) Marcar S(n) celdas (por ejemplo con el símbolo «X») en la cinta de trabajo 3.
2) Simular un paso de M en la cinta 1. Si M acepta, aceptar. Si M rechaza, rechazar.
3) Si $C = (4st)^{S(n)}$, rechazar.
4) Hacer C := C + 1 y volver a (2).

M' puede hacer (1) porque S(n) es espacio-construible. La comparación de (3) consiste en validar que la longitud de C coincide con la cantidad de marcas de la cinta de trabajo 3, y que todos los símbolos de C son el valor (4st – 1).

Se cumple que M' reconoce L y se detiene siempre:

Si M no «entra en loop», en (2) la MT M' acepta o rechaza como M. Si M «entra en loop», en (3) la MT M' rechaza porque detecta que M repitió una configuración.

También se cumple que M' trabaja en espacio S(n):

– En la cinta 1 se simula M, que trabaja en espacio S(n).
– El contador C de la cinta 2 escrito en base 4st no puede medir más que S(n), porque su valor máximo es $(4st)^{S(n)}$.
– La cinta 3 tiene S(n) marcas.

□

Ejercicio. *Probar el teorema anterior sin la asunción de que S(n) es una función espacio-construible.*

□

La prueba anterior reúne muchos elementos típicos de la complejidad espacial, que se verán en el próximo capítulo: se utilizan funciones espacio-construibles no menores que log(n), se acota el número de configuraciones distintas y se lleva la cuenta de las configuraciones recorridas, se marcan celdas para comparar longitudes, se representan números en una base adecuada, se lee el input sin traer a memoria fragmentos grandes del mismo, etc.
Por el Teorema de *Linear Tape Compression*, en las pruebas será indistinto tratar con MT que trabajen en espacio cS(n) o exactamente S(n). Además, excepto cuando amerite, en lugar de utilizar la cota $(4st)^{S(n)}$ se trabajará directamente con la expresión $c^{S(n)}$, siendo c una constante dependiente solamente de la MT de referencia.

CAPÍTULO 2.4. ESPACIO LOGARÍTMICO Y POLINOMIAL

Finalmente, en este último capítulo se profundiza sobre los problemas con resolución espacial logarítmica y polinomial. Además se establecen relaciones entre las jerarquías temporal y espacial, llegándose a definir una única jerarquía espacio-temporal.

En la Sección 2.4.1 se traza la frontera entre los problemas tratables y los problemas intratables, desde el punto de vista del espacio. En este caso, los problemas tratables son los que se pueden resolver con máquinas de Turing que trabajan en espacio logarítmico, lo que se justifica porque dichas máquinas no tardan más que un tiempo determinístico polinomial.

En la Sección 2.4.2 se demuestra mediante el Teorema de Savitch, que el no determinismo permite un ahorro de espacio del orden cuadrático con respecto al determinismo. Esto contrasta con lo que sucede con el tiempo, en que la relación es exponencial.

En la Sección 2.4.3 se trata nuevamente el concepto de completitud, esta vez aplicado a cualquier clase de problemas. Probar que un problema es completo en un clase, significa que es tanto o más complejo que cualquier otro de la clase, en el sentido espacial o temporal, según la medida de complejidad tenida en cuenta. Dado que las reducciones polinomiales en tiempo no son suficientes para establecer algunas relaciones de este tipo, se introducen las reducciones logarítmicas en espacio.

Entre otros, se destacan problemas relacionados con juegos entre un contrincante blanco y un contrincante negro, que se prueba que son PSPACE-completos con respecto a las reducciones polinomiales en tiempo. También se menciona un problema que se resuelve mínimamente en espacio exponencial. En el Apéndice 2 se sigue utilizando el concepto de completitud para relacionar jerarquías asociadas a distintos modelos de ejecución. Esto permite analizar, por ejemplo, qué problemas se podrían resolver de un modo más eficiente utilizando máquinas paralelas, máquinas de Turing probabilísticas, máquinas cuánticas, etc.

Finalmente, en la Sección 2.4.4 se prueba, mediante el Teorema de Immerman, que el espacio no determinístico es cerrado con respecto al complemento. Esto también contrasta con lo que sucede en el marco de la complejidad temporal. Por ejemplo, no se conoce la relación entre las clases temporales NP y CO-NP, y más aún, la conjetura de que NP $\neq$ CO-NP es más fuerte que la conjetura de que NP incluye estrictamente a P.

2.4.1. Problemas tratables

En esta sección se justifica por qué desde el punto de vista de la complejidad espacial, los problemas que se resuelven en espacio logarítmico son los que se consideran tratables.
Antes, cabe repasar un par de cotas superiores que relacionan espacio y tiempo, las cuales regirán lo que se tratará en ésta y las siguientes secciones:

- En tiempo T(n) no pueden recorrerse más de T(n) celdas. Esto corresponde al peor caso, en que siempre se va a la derecha o siempre se va a la izquierda. Como caso particular, tiempo determinístico T(n) puede acotarse en espacio determinístico del orden de $\sqrt{T(n)}$, cuando $T(n) \geq n^2$ y es tiempo-construible, $\sqrt{T(n)}$ es espacio-construible, y se consideran sólo MTD con una cinta.
- En espacio S(n) no pueden hacerse más de $c^{S(n)}$ pasos sin repetir alguna configuración, siendo c una constante que depende de la MT de referencia.

Un problema de DLOGSPACE es tratable, porque su solución tarda tiempo determinístico polinomial, como lo establece el siguiente teorema:

Teorema. DLOGSPACE $\subseteq$ P.

Prueba. Si una MTD M_1 trabaja en espacio O(log(n)), entonces existe una MTD M_2 equivalente que trabaja en tiempo $c^{\log(n)} = n^{\log(c)}$, siendo c una constante que depende de M_1.

□

Los problemas de DLOGSPACE se caracterizan por ser problemas de P con soluciones que consisten, básicamente, en validaciones sobre propiedades de componentes de las instancias de los problemas.
Un problema de NLOGSPACE también es tratable, porque se cumple NLOGSPACE $\subseteq$ P (se probará después). No se sabe si la inclusión es estricta, y por lo tanto tampoco se sabe si P incluye estrictamente a la clase DLOGSPACE.

Se prueba que $\bigcup_{k \geq 1}$ DSPACE($\log^k n$) $\neq$ P, pero no se sabe si un conjunto incluye al otro. También se prueba, por la jerarquía espacial, que DLOGSPACE $\subset$ DSPACE($\log^2 n$) $\subset$ DSPACE($\log^3 n$) $\subset$ …

Como $\cup_{k \geq 1}$ NSPACE($\log^k n$) = $\cup_{k \geq 1}$ DSPACE($\log^k n$), lo que se probará enseguida, entonces tampoco se puede establecer una relación de inclusión entre $\cup_{k \geq 1}$ NSPACE($\log^k n$) y P. También por la jerarquía espacial se cumple que NLOGSPACE $\subset$ NSPACE($\log^2 n$) $\subset$ NSPACE($\log^3 n$) $\subset$ ...

2.4.2. No determinismo vs determinismo

Se probó antes que la simulación determinística de una MTN que trabaja en tiempo T(n) tarda tiempo exponencial con respecto a T(n). ¿Cómo impacta, en cambio, el no determinismo en términos de espacio? Tomando como referencia dicha simulación, que consiste en ejecutar una a una las diferentes computaciones de la MTN, dada la posibilidad de reusar espacio podría suponerse que el impacto es menor. Efectivamente lo es, y se verá en esta sección cuánto.
Una primera aproximación podría ser que si una MTN M trabaja en espacio S(n), entonces existe una MTD M' equivalente que trabaja también en espacio S(n): como cada computación de M ocupa a lo sumo S(n) celdas, entonces M' simula toda computación de M en a lo sumo S(n) celdas, y al reusar espacio por cada computación, entonces el espacio total consumido termina siendo S(n). Pero esta prueba no es correcta, porque los discriminantes que debe generar M' para simular las computaciones de M miden $c^{S(n)} > S(n)$, dado que en S(n) celdas se pueden efectuar $c^{S(n)}$ pasos, siendo c una constante que depende de M.

Pasar del no determinismo al determinismo impacta un orden cuadrático en términos de espacio, como lo prueba el siguiente teorema:

Teorema de Savitch. Si $S(n) \geq \log_2 n$ es una función espacio-construible, y $L \varepsilon$ NSPACE(S(n)), entonces $L \varepsilon$ DSPACE($S^2(n)$).

Prueba. Sea L un lenguaje reconocido por una MTN M_1 que trabaja en espacio $S(n) \geq \log_2 n$, siendo S(n) espacio-construible.

Se cumple que en toda computación de M_1 a partir del input w, con |w| = n, existen a lo sumo $(4st)^{S(n)}$ configuraciones distintas, siendo $s = |Q_{M1}|$ y $t = |\Gamma_{M1}|$.

Sea $C_1 \vdash_M^i C_2$ la denotación de que la configuración C_2 de la MT M puede ser alcanzada desde la configuración C_1 de M en a lo sumo 2^i pasos. $C_1 \vdash_M C_2$ indica en particular que de C_1 se pasa a C_2 en un paso de M. Para $i \geq 1$, $C_1 \vdash_M^i C_2$ se puede validar generando una a una configuraciones C', y chequear si al menos en un caso vale $C_1 \vdash_M^{i-1} C'$ y $C' \vdash_M^{i-1} C_2$. El espacio necesario para determinar $C_1 \vdash_M^i C_2$ es, por lo tanto, el espacio necesario para almacenar las configuraciones C_1 y C_2 y una configuración auxiliar C', más el espacio necesario para validar $C_1 \vdash_M^{i-1} C'$ ó $C' \vdash_M^{i-1} C_2$ (esto es una disyunción y no una conjunción porque se puede reutilizar espacio).

Por ejemplo, para validar si M_1 pasa de C_1 a C_2 en $2^4 = 16$ pasos, se puede validar si pasa de C_1 a C^1 en $2^3 = 8$ pasos y de C^1 a C_2 en $2^3 = 8$ pasos, dada una configuración C^1. Para validar si se pasa de C_1 a C^1 en $2^3 = 8$ pasos, se puede validar si se pasa de C_1 a C^2 en $2^2 = 4$ pasos y de C^2 a C^1 en $2^2 = 4$ pasos. Para validar si se pasa de C_1 a C^2 en $2^2 = 4$ pasos, se puede validar si se pasa de C_1 a C^3 en $2^1 = 2$ pasos y de C^3 a C^2 en $2^1 = 2$ pasos. Y así sucesivamente. La idea general es que, reutilizando espacio cuando se puede y empleando una representación de datos adecuada, se puede validar que de C_1 se pasa a C_2 en 2^k pasos manteniendo en memoria mucho menos que 2^k ternas: con k ternas alcanza. El orden cuadrático indicado en el teorema sale del producto de k por el espacio requerido por cada terna.
Con estas consideraciones, se formula a continuación un algoritmo que corresponde a una MTD M_2 que simula M_1 en espacio $O(S^2(n))$. Dado el input w, con $n = |w|$ y $m = \lceil \log_2 4st \rceil$, M_2 trabaja de la siguiente manera (Test es una función a la que invoca el Programa Principal de M_2):

Programa Principal:
Sea C_0 la configuración inicial de M_1 con input w.
Por cada configuración final de aceptación C_f de M_1 hacer:
Si Test (C_0, C_f, mS(n)) = verdadero, entonces aceptar.
Rechazar.

Función Test (C_1, C_2, i):
Si (i = 0) y ($C_1 = C_2$ ó $C_1 \vdash_{M1} C_2$), entonces devolver el valor verdadero.
Si ($i \geq 1$), entonces por cada configuración C' de M_1 hacer:
Si Test (C_1, C', i – 1) y Test (C', C_2, i – 1), entonces devolver el valor verdadero.
Devolver el valor falso.

Una configuración de M_1 se representa por una 4-tupla (p_1, p_2, q, v): p_1 es un puntero en binario a la cinta de input de solo lectura (varía entre 1 y n+2), p_2 es un puntero en binario a la cinta de trabajo (varía entre 1 y S(n)), q es el estado corriente, y v es el contenido de la cinta de trabajo (string de S(n) símbolos). La configuración inicial C_0 es la 4-tupla (1, 1, q_0, BB…B). En una configuración final de aceptación C_f, el valor de q es q_A. Las distintas configuraciones C_f del programa principal de M_2, y las distintas configuraciones C' de la función Test, se van generando en el orden canónico.

M_2 puede generar el número S(n), S(n) símbolos blancos, un string de S(n) símbolos cualesquiera, etc., porque la función S(n) es espacio-construible. Además, se basa en la cota superior $(4st)^{S(n)}$ porque $S(n) \geq \log_2 n$.

Que se verifique en el programa principal la igualdad Test (C_0, C_f, mS(n)) = verdadero, para alguna configuración final de aceptación C_f, significa que M_1 acepta w en alguna de sus computaciones en a lo sumo $2^{mS(n)} = (4st)^{S(n)}$ pasos. Por eso sólo en este caso se acepta el input.

El parámetro i de la función Test varía entre 0 y mS(n), y se representa en binario. La validación $C_1 \vdash_{M1} C_2$ se lleva a cabo analizando C_1, C_2 y la relación de transición Δ de M_1. Se cumple Test (C_1, C_2, i) = falso, sólo cuando no existe ninguna configuración intermedia C' para la que vale $C_1 \vdash_{M1}^{i-1} C'$ y $C' \vdash_{M1}^{i-1} C_2$.

Por lo tanto, $L(M_2) = L(M_1)$.

Además, se cumple que la MTD M_2 trabaja en espacio $O(S^2(n))$. M_2 utiliza su cinta de trabajo como una pila:

– En cada invocación recursiva a la función Test, M_2 almacena C_1, C_2 e i como parámetros y C' como configuración local. Las representaciones de las configuraciones miden O(S(n)) celdas: el puntero a la cinta de input mide $\log_2(n+2)$, siendo $\log_2 n \leq S(n)$ (el input w no se pasa de invocación a invocación, su tamaño no se contabiliza en la complejidad espacial de M_2), el puntero a la cinta de trabajo mide $\log_2 S(n) \leq S(n)$, el estado corriente mide O(1), y el contenido de la cinta de trabajo mide S(n). El parámetro i de la función Test mide $\log_2(mS(n))$. En suma, cada invocación a la función Test ocupa O(S(n)) celdas.

– Como en cada invocación a la función Test, el parámetro i se decrementa en uno, y cuando se cumple i = 0 no hay invocación, entonces el número total de invocaciones es (mS(n) + 1) = O(S(n)).
– Por lo tanto, el espacio total requerido por la MTD M_2 es O(S(n))**.**O(S(n)) = O(S^2(n)) celdas.

□

Notar que por el Teorema de Savitch, vale para todo k ≥ 1: NSPACE (n^k) ⊆ DSPACE(n^{2k}), es decir que NPSPACE ⊆ PSPACE. Como por definición también vale PSPACE ⊆ NPSPACE, entonces se puede formular lo siguiente:

Corolario. PSPACE = NPSPACE.

□

Así, a diferencia de lo que sucede con el tiempo, en la complejidad espacial la clase de problemas con solución no determinística polinomial coincide con la clase de problemas con solución determinística polinomial.

La figura 2.4.1 presenta lo que en este libro se identificará como el *mapa de la complejidad temporal-espacial*, asumiendo que todas las inclusiones son estrictas:

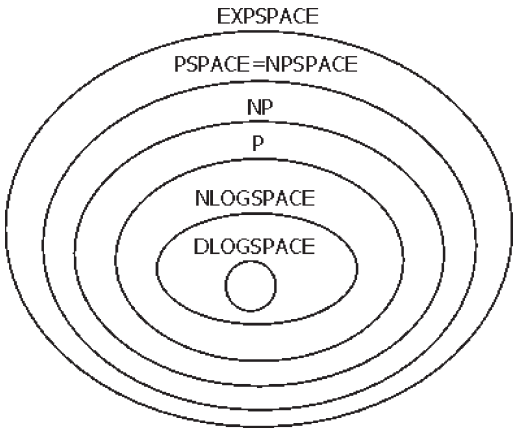


Figura 2.4.1

La prueba de que NLOGSPACE $\subseteq$ P se verá en la próxima sección.

Se cumple que NP $\subseteq$ PSPACE porque NP $\subseteq$ NPSPACE (en cn^k pasos no pueden recorrerse más de cn^k celdas) y NPSPACE = PSPACE. Otra forma de probar lo mismo es utilizando la simulación de siempre de una MTN M por una MTD M':

- Cada computación de M tarda cn^k pasos, y por lo tanto ocupa a lo sumo cn^k celdas. M' reusa el mismo espacio polinomial por cada computación simulada.
- Los discriminantes de M' son strings de cn^k dígitos entre 1 y K, siendo K el grado de la relación de transición Δ_M.

Dadas las inclusiones:

DLOGSPACE $\subseteq$ NLOGSPACE $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE,

es un problema abierto determinar si las inclusiones son estrictas. Es interesante observar que por la jerarquía espacial se cumple que DLOGSPACE $\subset$ PSPACE, por lo que al menos una inclusión intermedia es estricta.

2.4.3. Problemas completos

El concepto de NP-completitud tratado en la complejidad temporal es un caso particular del concepto de completitud en cualquier clase C de problemas, temporal o espacial.
Para el caso de NP se estableció que probar que un problema es NP-completo, con respecto a las reducciones polinomiales en tiempo, en la práctica es probar que no pertenece a P, a menos que P = NP. El mismo criterio es aplicable a cualquier clase C, haciendo referencia a un tipo determinado de reducción. Hasta ahora se utilizaron solamente las reducciones polinomiales en tiempo. Se define a continuación otro tipo de reducción que será necesario considerar.

Definición. Sean L_1 y L_2 dos lenguajes incluidos en Σ^*. Existe una *reducción logarítmica en espacio* de L_1 a L_2 , sii existe una reducción de L_1 a L_2 y la función de reducción f es computable por una MTD M que trabaja en espacio O(log(n)).

□

Dado un input w, el output f(w) generado se escribe en la cinta de output de M_f. Su tamaño no interviene en el cálculo del espacio en que trabaja la MT (lo mismo que sucede con el tamaño del input).

Para simplificar la escritura, de ahora en más las reducciones polinomiales en tiempo y las reducciones logarítmicas en espacio se referirán también como reducciones *poly-time* y *log-space*, respectivamente. $L_1 \alpha_P L_2$ también podrá escribirse $L_1 \alpha_{\text{poly-time}} L_2$. La expresión $L_1 \alpha_{\text{log-space}} L_2$ denotará que existe una reducción log-space de L_1 a L_2. Además, $f \varepsilon$ DLOGSPACE (podría usarse también $f \varepsilon$ FDLOGSPACE) indicará que la función f se computa en espacio determinístico logarítmico.

Se prueba fácilmente que toda reducción log-space es una reducción poly-time (queda como ejercicio para el lector).

Ejemplo. La reducción utilizada en el Teorema de Cook para probar que el lenguaje SAT es NP-completo es log-space, y por lo tanto poly-time:

Dado el input w, con |w| = n, el output E_w, que es una fórmula booleana, mide $O(p(n)^2)$, siendo p(n) un polinomio. E_w es lo suficientemente simple como para que la MTD M_f que computa la función de reducción sólo haga uso fundamentalmente del espacio necesario para contar hasta $(p(n)+1)^2$, que es O(log(n)).

El siguiente teorema fundamenta cómo utilizar las reducciones log-space para poblar la clase DLOGSPACE. En realidad, las reducciones log-space (y poly-time) se aplicarán a un espectro mucho más amplio de clases. En el mismo teorema se resuelve una dificultad técnica que no apareció en el caso de la complejidad temporal:

Teorema. Si $L_1 \alpha_{\text{log-space}} L_2$ y $L_2 \varepsilon$ DLOGSPACE, entonces $L_1 \varepsilon$ DLOGSPACE.

Prueba. No sirve construir una MTD M_1 que componga directamente la MTD M_f que reduce L_1 a L_2 en espacio logarítmico con la MTD M_2 que reconoce L_2 en espacio logarítmico: dado el input w, con |w| = n, el

output de M_f es un string f(w) que puede medir $n^k > O(\log(n))$, con k constante.
Este problema se resuelve del siguiente modo. La idea es que M_f no escriba todo el string f(w), sino que le aporte a M_2, cada vez, un solo símbolo, el que M_2 requiera: mientras M_2 se mueva a la derecha en su cinta de input, M_f le brindará el siguiente símbolo, y cuando M_2 se mueva a la izquierda, M_f arrancará desde el comienzo y se detendrá cuando llegue a producir el símbolo requerido.
Formalmente, la MTD M_1 que se va a construir para reconocer el lenguaje L_1 en espacio logarítmico, trabaja de la siguiente manera:
Una cinta de trabajo de M_1 contiene un contador con la posición del cabezal al input de M_2 (es decir al output de M_f), escrito en binario e inicializado en uno. Como este string no tiene más de n^K símbolos, entonces el contador no va a ocupar más de $k(\log_2 n)$ celdas. Las otras cintas de trabajo de M_1 se utilizan para simular las MTD M_f y M_2.
M_1 empieza simulando M_f hasta que genera el primer símbolo del output, y luego simula el primer paso de M_2. Luego itera del siguiente modo. Por cada movimiento simulado de M_2 que mueve el cabezal de su input a la derecha, M_1 suma uno al contador, simula M_f hasta que genera el siguiente símbolo (si M_f se detiene antes se asume que el símbolo es «B»), y simula el siguiente paso de M_2. Por cada movimiento simulado de M_2 que mueve el cabezal de su input a la izquierda, M_1 resta uno al contador, reinicia la simulación de M_f desde el comienzo hasta que genera el símbolo i requerido, siendo i el contenido del contador (si i = 0 se asume que el símbolo es «B»), y simula el siguiente paso de M_2. Y mientras M_2 no mueva su cabezal al input, M_1 la irá simulando paso a paso.

Se comprueba fácilmente que M_1 reconoce el lenguaje L_1. Además, M_1 trabaja en espacio determinístico logarítmico porque:

- La MTD M_f trabaja en espacio logarítmico a partir de un input w, con $|w| = n$.
- La MTD M_2 trabaja en espacio logarítmico a partir del output f(w) de M_f, siendo $|f(w)| \leq n^k$, con k constante, por lo que ocupa O(log(n)) celdas.

□

Como en el caso de las reducciones poly-time, se cumple que la relación $\alpha_{\text{log-space}}$ es reflexiva y transitiva. También se prueba que las clases DLOGSPACE, NLOGSPACE, P, NP y PSPACE son cerradas con

respecto a las reducciones log-space, es decir, en cada una de estas clases C se cumple que si L $\alpha_{\text{log-space}}$ L' y L' ε C, entonces L ε C.

Ejercicio. *Probar que la relación $\alpha_{log\text{-}space}$ es reflexiva y transitiva, y que DLOGSPACE, NLOGSPACE, P, NP y PSPACE son cerradas con respecto a las reducciones log-space.*

□

Utilizando reducciones poly-time y log-space, se puede extender el alcance del concepto de completitud a cualquier clase de lenguajes C:

Definición. Sea C una clase de lenguajes. L es un lenguaje *C-difícil* con respecto a las reducciones poly-time (respectivamente log-space) sii para todo lenguaje L' ε C se cumple que L' $\alpha_{\text{poly-time}}$ L (respectivamente L' $\alpha_{\text{log-space}}$ L). Si L ε C, entonces L es un lenguaje *C-completo*.

□

Con las consideraciones anteriores, entonces una manera de resolver la relación entre las clases DLOGSPACE, NLOGSPACE, P, NP y PSPACE, es la que se enuncia en el teorema que se presenta a continuación. El razonamiento es el mismo que se empleó particularmente para analizar la relación entre P y NP: ante la sospecha de que $C_1 \subset C_2$, probar que un problema es C_2-completo, en la práctica es demostrar que no pertenece a C_1, a menos que $C_1 = C_2$.

Teorema. Se cumple que:

a) Si L es NLOGSPACE-completo con respecto a las reducciones log-space, y L está en DLOGSPACE, entonces DLOGSPACE = NLOGSPACE.

b) Si L es P-completo con respecto a las reducciones log-space, y L está en DLOGSPACE, entonces DLOGSPACE = P.

c) Si L es P-completo con respecto a las reducciones log-space, y L está en NLOGSPACE, entonces NLOGSPACE = P.

d) Si L es PSPACE-completo con respecto a las reducciones poly-time, y L está en P, entonces P = PSPACE.

e) Si L es PSPACE-completo con respecto a las reducciones poly-time, y L está en NP, entonces NP = PSPACE.

Ejercicio. *Probar el teorema.*

□

Se pueden plantear más incisos al teorema anterior, incluyendo por ejemplo a las clases EXP y EXPSPACE. La C-completitud también se puede utilizar para probar la igualdad de dos clases, como se establece en el siguiente teorema:

Teorema. Dada las clases de lenguajes C_1 y C_2, si:

a) un lenguaje L es C_1-completo y C_2-completo con respecto a las reducciones log-space (respectivamente poly-time), y
b) las clases C_1 y C_2 son cerradas con respecto a las reducciones log-space (respectivamente poly-time),

entonces $C_1 = C_2$.

Ejercicio. *Probar el teorema.*

□

El estudio de los problemas completos es muy importante en la complejidad computacional, porque capturan la esencia de la dificultad de una clase de complejidad. De hecho, una clase adquiere trascendencia cuando incluye problemas interesantes que se prueban completos en la clase. Tal es el caso, entre otras, de las clases NLOGSPACE, P, NP y PSPACE.
El análisis va más allá de la complejidad logarítmica, polinomial o exponencial. Por ejemplo, si un lenguaje L fuera NP-completo con respecto a las reducciones log-space, y L perteneciera a $DTIME(n^{\log(n)})$, entonces se cumpliría $NP \subseteq DTIME(n^{c\log(n)})$, para alguna constante c (en general, si L perteneciera a DTIME(T(n)), valdría $NP \subseteq DTIME(T(n^c))$, para alguna constante c).
Para completar esta sección, se enumeran a continuación algunos ejemplos de problemas C-completos. El primer ejemplo retoma el caso del lenguaje CON, que representa el problema de la alcanzabilidad. Se había probado que CON ε NLOGSPACE. Ahora se va a probar que CON es NLOGSPACE-completo con respecto a las reducciones log-

space. Después se verá cómo esto se vincula con la relación entre las clases NLOGSPACE y P.

Ejemplo. Sea CON = {(G, s, t) | G es un grafo orientado, *s* y *t* son el primero y último vértice de G, y existe un camino en G desde s hasta t}. Se prueba que CON es NLOGSPACE-completo con respecto a las reducciones log-space. Se construirá una reducción log-space al lenguaje CON desde cualquier lenguaje L ε NLOGSPACE.

Dado L ε NLOGSPACE, sea M una MTN que lo reconoce en espacio logarítmico. Toda configuración C de M se puede representar en O(log(n)) celdas: el cabezal a la cinta de input, el cabezal a la cinta de trabajo y el contenido de la cinta de trabajo se pueden representar en O(log(n)) celdas, mientras que el estado corriente ocupa espacio constante. Se va a construir una MTD M' que trabaja en espacio O(log(n)) y transforma un input w de Σ* en un grafo orientado G_w, con un camino del primero al último vértice sii M acepta w. Los vértices de G_w representan las configuraciones de M. G_w tiene además un vértice especial final v. El primer vértice representa la configuración inicial de M con input w (cabezales con valor uno, contenido con «B», y estado corriente q_0).

La MTD M' trabaja de la siguiente manera:

1) Primero, escribir la configuración inicial de M con input w como primer vértice. Después, escribir el resto de las configuraciones de M como siguientes vértices, según el orden canónico, sin repetir el primer vértice. Finalmente, escribir el vértice especial v.
2) Sea C la primera configuración de M según el orden canónico.
3) Si C no es una configuración final (de aceptación o rechazo), obtener todas las configuraciones D tales que $C|\!-_M D$, y escribir todos los arcos (C, D). Si en cambio C es una configuración final de aceptación, escribir el arco (C, v).
4) Si C es la última configuración de M según el orden canónico, detenerse. Si no, sea C la siguiente configuración de M según el orden canónico, y volver a (3).

Se comprueba fácilmente que G_w tiene un camino del primero al último vértice sii w ε L(M). Además, la MTD M' trabaja en espacio O(log(n)) porque mantiene en memoria unas pocas configuraciones, cada una

de las cuales mide O(log(n)). Para la generación de las configuraciones contribuye el hecho de que la función log(n) es espacio-construible.

Se prueba que CON ε P: se puede utilizar la técnica *depth first search* para recorrer el grafo en tiempo determinístico polinomial. De esta manera se prueba que NLOGSPACE $\subseteq$ P:

Dado L ε NLOGSPACE, entonces L $\alpha_{\text{log-space}}$ CON, y por lo tanto L $\alpha_{\text{poly-time}}$ CON. Como CON ε P, entonces por ser P cerrada con respecto a las reducciones poly-time, se cumple que L ε P.

El siguiente ejemplo introduce un lenguaje que es una generalización del lenguaje SAT. Representa el problema de la satisfactibilidad de las fórmulas booleanas con cuantificadores.

Ejemplo. Sea el lenguaje QBF = {ϑ | ϑ es una fórmula booleana con cuantificadores, no tiene variables libres y es verdadera}. Se cumple que QBF es PSPACE-completo con respecto a las reducciones poly-time.

Notar que una fórmula booleana E sin cuantificadores y con variables $x_1, x_2, x_3, \ldots, x_k$, es satisfactible, sii la fórmula booleana $\exists x_1 \exists x_2 \exists x_3 \ldots \exists x_k(E)$ es verdadera, por lo que el lenguaje SAT es un caso particular de QBF, y así QBF es NP-difícil con respecto a las reducciones poly-time.
No se conoce solución no determinística polinomial en tiempo para este problema. Por ejemplo, dada la fórmula $\vartheta = \forall x \forall y \forall z\ (x \wedge y \wedge z)$, no alcanza con chequear que una asignación de valores de verdad para la terna (x, y, z) sea verdadera, sino que hay validar que las 2^3 posibilidades de valores verdadero y falso para x, y, z, resulten verdaderas.
Sea este otro ejemplo: $\vartheta = \forall x[\forall y[\exists z\ (x \vee y \vee z)]]$. La evaluación de ϑ se efectúa así (para simplificar la escritura, se utiliza 1 por verdadero y 0 por falso):

1) Primero, se consideran los valores de verdad de x: ($\forall y[\exists z\ (1 \vee y \vee z)]) \wedge (\forall y[\exists z\ (0 \vee y \vee z)])$.
2) Luego, de y : $(\exists z\ (1 \vee 1 \vee z)) \wedge (\exists z\ (1 \vee 0 \vee z)) \wedge (\exists z\ (0 \vee 1 \vee z)) \wedge (\exists z\ (0 \vee 0 \vee z))$.
3) Finalmente, de z : $[(1 \vee 1 \vee 1) \vee (1 \vee 1 \vee 0)] \wedge [(1 \vee 0 \vee 1) \vee (1 \vee 0 \vee 0)] \wedge [(0 \vee 1 \vee 1) \vee (0 \vee 1 \vee 0)] \wedge [(0 \vee 0 \vee 1) \vee (0 \vee 0 \vee 0)] = 1$.

La idea general de la prueba de que QBF ε PSPACE es la siguiente. Sea EVAL una función recursiva que trabaja de la siguiente manera:

1) EVAL(0) = 0.
2) EVAL(1) = 1.
3) EVAL(E, $\neg$) = $\neg$ EVAL(E).
4) EVAL(E_1, E_2, $\vee$) = EVAL(E_1) $\vee$ EVAL(E_2).
5) EVAL(E_1, E_2, $\wedge$) = EVAL(E_1) $\wedge$ EVAL(E_2).
6) EVAL(E, $\forall$x) = EVAL(E^x_1) $\wedge$ EVAL(E^x_0).
7) EVAL(E, $\exists$ x) = EVAL(E^x_1) $\vee$ EVAL(E^x_0).

La expresión E^x_i denota la sustitución de x por i en E. Como el número de cuantificadores más operadores de ϑ es a lo sumo $|\vartheta|$ = n, entonces la profundidad de la recursión es a lo sumo n. Además, en cada invocación recursiva los parámetros miden O(n). Por lo tanto, el espacio determinístico requerido es $O(n^2)$.
Se demuestra que QBF es PSPACE-difícil con respecto a las reducciones poly-time, construyendo una reducción poly-time desde cualquier lenguaje de PSPACE a QBF. La prueba se basa en representar configuraciones de MT como en la prueba de pertenencia de SAT a NPC (no se desarrollará).

Tal como se hizo con los lenguajes NP-completos, en que luego de encontrar un primer lenguaje de la clase, el lenguaje SAT, se fueron encontrando otros lenguajes NP-completos por medio de las reducciones poly-time, esto aplica en cualquier clase C y con los dos tipos de reducciones que se definieron.
El ejemplo siguiente se basa en este mecanismo. Se refiere a otra instanciación, como 3-SAT, del lenguaje K-SAT de las fórmulas booleanas satisfactibles que tienen forma normal conjuntiva y K literales por cláusula.

Ejemplo. Sea 2-SAT el lenguaje de las fórmulas booleanas satisfactibles que tienen forma normal conjuntiva y 2 literales por cláusula.

Se probó que 3-SAT es NP-completo con respecto a las reducciones poly-time. Para el caso de 2-SAT, en cambio, se prueba que el lenguaje

está en P. Más aún, 2-SAT es NLOGSPACE-completo con respecto a las reducciones log-space. El esquema de la prueba es el siguiente:

- Se prueba que 2-SAT ε NLOGSPACE.
- Se probó antes que CON es NLOGSPACE-completo con respecto a las reducciones log-space. NLOGSPACE es cerrado con respecto a la operación complemento (en realidad, como se probará en la siguiente sección, NSPACE(S(n)) = CO-NSPACE(S(n)) para toda función S(n) espacio-construible no menor que log(n)). Por lo tanto, CON^C es NLOGSPACE-completo con respecto a las reducciones log-space.
- Se prueba que CON^C $\alpha_{\text{log-space}}$ 2-SAT.
- Finalmente, 2-SAT es NLOGSPACE-completo con respecto a las reducciones log-space.

Notar, tal como se usó en el esquema de la prueba, que siendo la clase NLOGSPACE cerrada con respecto al complemento, se la puede «poblar» con lenguajes completos tales que sus complementos son completos (queda como ejercicio para el lector probar la correctitud de este mecanismo).

El ejemplo siguiente introduce el problema de evaluación de circuitos, que también será referenciado en el Apéndice 2 cuando se traten las máquinas paralelas.

Ejemplo. Dado un circuito booleano c de (n + q) nodos, con n nodos de input, el problema de evaluación de circuitos consiste en decidir, dado un input w de longitud n, si c(w) = verdadero.

Se prueba que este problema es P-completo con respecto a las reducciones log-space. Si CIRCUIT VALUE es el lenguaje asociado al problema, se puede construir una reducción log-space desde cualquier lenguaje de P a CIRCUIT VALUE, para probar que CIRCUIT VALUE es P-difícil.

A partir del ejemplo anterior, se puede probar que el problema de la programación lineal entera también es P-completo con respecto a las reducciones log-space:

Ejemplo. El problema de la programación lineal entera se define de la siguiente manera: «Dada una matriz entera A de $n \times m$, un vector entero B de tamaño n, un vector entero C de tamaño m, y un entero k, ¿existe un vector racional no negativo X de tamaño m que satisface $AX \leq B$ y $CX \geq k$?» INTEGER LINEAR PROGRAMMING es el lenguaje asociado al problema.
Se cumple que este problema es P-completo con respecto a las reducciones log-space. Para probar que es P-difícil, se construye una reducción log-space de CIRCUIT VALUE a INTEGER LINEAR PROGRAMMING.

Distintos problemas que genéricamente se conocen como *juegos entre blanco y negro*, se prueba que son PSPACE-completos con respecto a las reducciones poly-time.
Dichos problemas se especifican de la siguiente manera:

- Existe una configuración inicial y una configuración final.
- El blanco y el negro alternan jugadas uno tras otro, impactando sobre la configuración corriente. La primera jugada es del blanco.
- El problema consiste en decidir si el blanco gana en n movimientos, o en otras palabras, si existe una jugada del blanco de la configuración C_0 a la configuración C_1 tal que para todas las jugadas del negro de C_1 a C_2 existe una jugada del blanco de C_2 a C_3 tal que…existe una jugada del blanco de C_{n-2} a C_{n-1} tal que para todas las jugadas del negro de C_{n-1} a C_n la configuración C_n es ganadora para el blanco.

En esta gama de juegos se incluyen el juego de geografía y el go:

Ejemplo. El juego de geografía se representa por el lenguaje GEO. El juego consiste en lo siguiente: el blanco empieza nombrando una ciudad, el negro sigue nombrando otra ciudad que empiece con la letra con la que terminó la ciudad anterior (no se pueden repetir ciudades), luego

sigue el blanco de la misma manera, luego el negro, etc., hasta que un jugador no tiene más ciudades para nombrar.
Se prueba que GEO ε PSPACE, y que existe una reducción poly-time de QBF a GEO, por lo que GEO es PSPACE-completo con respecto a las reducciones poly-time.
Otro juego PSPACE-completo con respecto a las reducciones poly-time dentro de esta gama es el go, representado por el lenguaje GO. Se puede construir una reducción poly-time de GEO a GO.

El último ejemplo se refiere a un problema intratable ya referido previamente:

Ejemplo. El lenguaje REX representa el problema de decidir si una expresión regular con exponenciación denota todos los strings sobre un alfabeto.
Se prueba que el lenguaje REX es EXPSPACE-completo con respecto a las reducciones poly-time.
Se puede construir una reducción poly-time a REX desde cualquier lenguaje de DSPACE($2^{p(n)}$), siendo p(n) un polinomio.

Notar que como en el caso de P con respecto a las reducciones poly-time, todos los lenguajes de DLOGSPACE son DLOGSPACE-completos con respecto a las reducciones log-space (la prueba es similar).
Por lo tanto, para encontrar los lenguajes más difíciles de DLOGSPACE se debería recurrir a otro tipo de reducción.

La figura 2.4.2 vuelve a presentar el mapa de la complejidad temporal-espacial, pero ahora «poblado» con algunos de los problemas C-completos mencionados previamente:

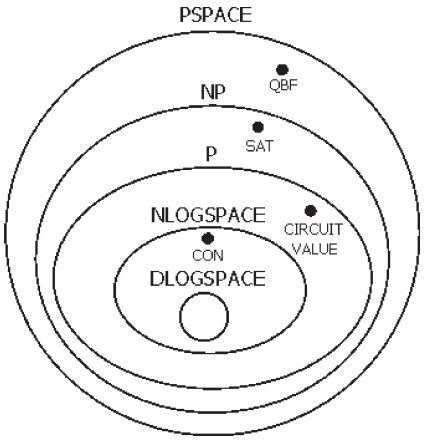


Figura 2.4.2

2.4.4. No determinismo y complemento

En esta última sección se prueba que toda clase espacial no determinística es cerrada con respecto a la operación de complemento. Recordar que en el marco de la complejidad temporal no se conoce la relación entre NTIME(T(n)) y CO-NTIME(T(n)), que es la clase con los lenguajes complemento de los que integran NTIME(T(n)). En particular, se probó que la conjetura NP $\neq$ CO-NP es más fuerte que la conjetura P $\neq$ NP.

Teorema de Immerman. Para toda función espacio-construible S(n) $\geq$ log(n), se cumple que NSPACE(S(n)) = CO-NSPACE(S(n)).

Prueba. Simular determinísticamente una MTN M que trabaja en espacio S(n) reusando el espacio por cada computación no sirve, porque el discriminante que indica en cada paso cómo seguir según la relación Δ_M mide $c^{S(n)}$, siendo c una constante que depende de M.

Se enunciarán y probarán dos lemas que constituyen la prueba del teorema.

Lema 1. Sea $S(n) \geq \log(n)$ una función espacio-construible, M una MTN que trabaja en espacio S(n), START la configuración inicial de M dado un input w, y N el número exacto de configuraciones alcanzables por M desde START. Existe una MTN M' que trabaja en espacio S(n) que acepta el complemento de L(M).

Lema 2. Con las mismas hipótesis del lema anterior, existe una MTN M'' que calcula N en espacio S(n).

Antes de probar los dos lemas, se muestra cómo a partir de ellos se demuestra el teorema:

- M'' es una MTN que calcula N en espacio S(n). Se asume que M'' calcula el mismo output N en todas sus computaciones que aceptan (en las que rechazan no hay output).
- M' es una MTN que dado N reconoce el complemento de L(M) en espacio S(n).
- Sea M^C una MTN que compone M'' con M'. M^C reconoce el complemento de L(M) dado que M' lleva a cabo esta tarea. Y M^C trabaja en espacio S(n) porque M'' y M' trabajan en espacio S(n), y el número N en base 4st no mide más de S(n) celdas, siendo $s = |Q_M|$ y $t = |\Gamma_M|$.

Prueba del Lema 1. Se va a construir la MTN M'. En la cinta de input de solo lectura está el input w. En una cinta de trabajo está el valor N. En otra cinta, M' genera en orden canónico una a una las configuraciones C de M a partir de w, con la representación habitual (miden O(S(n))). En otra cinta se mantiene un contador D inicializado en cero, para llevar la cuenta de las configuraciones generadas (al igual que N, representado en base 4st mide a lo sumo S(n)). Y en las otras cintas simula M.

M' trabaja de la siguiente manera:

1) Generar la primera configuración C de M con input w según el orden canónico.
2) Simular M. Si se alcanza C y C es una configuración de aceptación, rechazar. Si en cambio se alcanza C y C no es una configuración de aceptación, hacer D := D + 1.
3) Si D = N, aceptar.

4) Si C es la última configuración de M con input w según el orden canónico, rechazar. En caso contrario, generar la siguiente configuración C de M con input w según el orden canónico, y volver a (2).

M' reconoce el complemento de L(M):

- Si M acepta w, entonces hay a lo sumo (N – 1) configuraciones C alcanzables por M desde START que no son de aceptación. Así, toda computación de M' o bien rechaza en (2), si es que encuentra una configuración de aceptación, o bien rechaza en (4), si es que no detectó las N configuraciones alcanzables desde START.
- Si M rechaza w, entonces hay N configuraciones C alcanzables por M desde START que no son de aceptación. Así, toda computación de M' o bien acepta en (3), si es que detectó las N configuraciones alcanzables desde START, o bien rechaza en (4), si es que no detectó las N configuraciones, pero al menos en una computación M' acepta.

M' trabaja en espacio S(n):

M trabaja en espacio S(n), las configuraciones C miden O(S(n)), y N y el contador D miden a lo sumo S(n). Por el Teorema de *Linear Tape Compression*, existe una MTN equivalente a M' que trabaja en espacio S(n), que para simplificar la escritura se seguirá denominando M'.

Prueba del Lema 2. Se presenta a continuación una idea general de cómo trabaja la MTN M'':

Sea N_d el número de configuraciones C alcanzables por M desde START en a lo sumo d pasos. M'' calcula primero N_1, luego N_2, luego N_3, y así sucesivamente, hasta que en un momento dado se cumple $N_{k-1} = N_k = N$. La obtención de N_1 es trivial. En lo que sigue, se muestra cómo M'' calcula N_{d+1} a partir de N_d, para $d \geq 1$:

En la cinta de input de solo lectura está el input w. Entre otras cintas de trabajo utilizadas, en una cinta la MT M'' genera en orden canónico una a una las configuraciones C de M a partir de w, en otra cinta mantiene un contador D inicializado en cero para llevar la cuenta de las configuraciones alcanzables en d+1 pasos, y en otras cintas simula M. En la cinta de output, la M'' genera el número N.

La MTN M'' calcula N_{d+1} a partir de N_d de la siguiente manera. Por cada configuración C según el orden canónico, simula a lo sumo d pasos de M (si al cabo de todas las simulaciones no detecta las N_d configuraciones alcanzables, rechaza). Por cada una de las N_d instancias detectadas, chequea si es igual a C o si C es alcanzable en un paso, y si se cumple hace D := D + 1.

M'' trabaja en espacio O(S(n)) porque M trabaja en espacio S(n), las configuraciones C miden O(S(n)), y *d* y el contador D miden en base 4st a lo sumo S(n). Por el Teorema de *Linear Tape Compression* existe una MTN equivalente a M'' que trabaja en espacio S(n), que para simplificar la escritura se seguirá denominando M''.

□

Apéndice 2

En este apéndice se trata la complejidad computacional de problemas más generales que los de decisión (Capítulo A2.1). Se describen además, suscintamente, modelos de máquinas alternativas a las máquinas de Turing, y se analiza cómo las jerarquías de clases asociadas se relacionan con las que se definieron en los capítulos precedentes (Capítulos A2.2 a A2.5). El apéndice se organiza de la siguiente manera:

En el Capítulo A2.1 se presentan los problemas de búsqueda y enumeración. En la Sección A2.1.1 se tratan los problemas de optimización, y en la Sección A2.1.2, los problemas #P. En el Capítulo A2.2 se presentan las máquinas de Turing con oráculo. En el Capítulo A2.3, las máquinas de Turing probabilísticas. En el Capítulo A2.4, las máquinas para paralelismo. Finalmente, en el Capítulo A2.5 se presentan los sistemas de pruebas interactivas (Sección A2.5.1), las máquinas de Turing alternantes (Sección A2.5.2), y las máquinas cuánticas (Sección A2.5.3).

A2.1. Problemas de búsqueda y enumeración

A diferencia de los problemas de decisión, resolver un *problema de búsqueda* significa encontrar una de las soluciones del problema (por ejemplo, uno de los circuitos de Hamilton de un grafo), o bien responder negativamente si no hay solución.
De todos modos, tratar con problemas de decisión aporta valiosa información para los problemas de búsqueda asociados. Por ejemplo, buscar en un grafo G un circuito de Hamilton con costo mínimo no puede tardar menos que decidir si en G existe un circuito de Hamilton con costo $\leq$ B. En particular, el problema de búsqueda no debería poder resolverse en tiempo determinístico polinomial siendo el problema de decisión asociado NP-completo, porque si no, éste también se resolvería en tiempo determinístico polinomial de la siguiente manera:

1) Encontrar un circuito de Hamilton de G de costo mínimo C, y si no existe rechazar (esto tarda tiempo determinístico polinomial).
2) Aceptar si C $\leq$ B, y en caso contrario rechazar (esto también tarda tiempo determinístico polinomial).

Lo mismo ocurre con un tercer tipo de problemas, los *problemas de enumeración*, en los que se debe calcular el número de soluciones. Por ejemplo, si calcular la cantidad de circuitos de Hamilton que tiene un grafo G tarda tiempo determinístico polinomial, lo mismo tardaría decidir si G tiene un circuito de Hamilton (que es un problema NP-completo):

1) Calcular la cantidad N de circuitos de Hamilton de G (esto tarda tiempo determinístico polinomial).
2) Aceptar si $N \geq 0$, y en caso contrario rechazar (esto también tarda tiempo determinístico polinomial).

Los *problemas de optimización* son un caso particular de los problemas de búsqueda, y se describen a continuación (Sección A2.1.1), con foco en la complejidad temporal. En la Sección A2.1.2 se hace referencia a una clase de los problemas de enumeración.

A2.1.1. Problemas de optimización

Los problemas de optimización consisten en la búsqueda de una solución de valor máximo (*maximización*) o mínimo (*minimización*). Dada una instancia de problema x, y una posible solución y, se asume la existencia de una función m que se denomina *medida de la solución*, tal que m(x, y) es un número entero menor o igual o mayor o igual que la *solución óptima* opt(x), según se busque un máximo o un mínimo, respectivamente.
Una primera categorización de los problemas de optimización se establece en términos de dos clases de complejidad computacional, PO y NPO (la segunda incluye a la primera), que se corresponden con P y NP, respectivamente.
Un problema de maximización o minimización de PO se resuelve en tiempo determinístico polinomial. La definición de los problemas de NPO es una generalización de la definición de los problemas de NP:

a) El conjunto de instancias de problema x pertenece a P.
b) Las posibles soluciones *y* son tales que $|y| \leq p(|x|)$, siendo p un polinomio, y siendo decidible en tiempo determinístico polinomial si un string *y*, con $|y| \leq p(|x|)$, es una posible solución.
c) Es decidible en tiempo determinístico polinomial si *y* es solución de x.
d) m(x, y) se calcula en tiempo determinístico polinomial.

A todo problema de optimización de PO y NPO se le asocia un lenguaje o problema de decisión, tal como se muestra en el siguiente ejemplo:

Ejemplo. Sea el problema de encontrar un cubrimiento de vértices de tamaño mínimo de un grafo. Su lenguaje asociado es:

minVC = {(G, K) | el grafo G tiene un cubrimiento de vértices de tamaño a lo sumo K}.

Sea por otro lado el problema de encontrar un clique de tamaño máximo de un grafo. Su lenguaje asociado es:

maxCLIQUE = {(G, K) | el grafo G tiene un clique de tamaño al menos K}.

Es decir, a un problema de maximización (respectivamente minimización) se le asocia el problema de decidir si existe una solución que mide al menos (respectivamente a lo sumo) un valor K, teniendo en cuenta que la complejidad del problema de optimización no puede ser menor que la complejidad del problema de decisión asociado.
Se prueba fácilmente que el lenguaje asociado a un problema de PO (respectivamente NPO) pertenece a P (respectivamente NP). Obviamente, si $P \neq NP$, entonces todo problema de NPO cuyo lenguaje asociado es NP-completo no está en PO, es decir, es un problema de optimización intratable. De este modo, si $P \neq NP$ entonces $PO \neq NPO$. Asumiendo la conjetura $PO \neq NPO$, se utilizan *aproximaciones polinomiales*. La idea es, en los casos de problemas de NPO con lenguaje asociado NP-completo, sacrificar la búsqueda del óptimo y construir un algoritmo eficiente que produzca una solución «buena», cercana al óptimo según un criterio determinado. El siguiente es un ejemplo de aproximación polinomial:

Ejemplo. Se busca una aproximación polinomial del problema de búsqueda de un cubrimiento de vértices mínimo de un grafo (el lenguaje asociado es NP-completo).
Dado un grafo G = (V, E), la MTD M siguiente genera en tiempo polinomial un cubrimiento de vértices $V' \subseteq V$ de G:

1) Hacer V' := $\varnothing$ y E' := E.
2) Si E' = $\varnothing$, aceptar.
3) Hacer E' := E' – {(v_1, v_2)}, siendo (v_1, v_2) un arco de E'.
4) Si $v_1 \notin$ V' y $v_2 \notin$ V', entonces hacer V' := V' $\cup$ {v_1, v_2}.
5) Volver a (2).

V' cubre un conjunto A de arcos disjuntos de G, y |A| = |V'| / 2 porque los dos vértices de cada uno de los arcos de A están en V'.

Cualquier cubrimiento de vértices C de G debe tocar por definición todos los arcos de A, y por lo tanto debe medir al menos |C| = |V'| / 2.

De este modo, el tamaño del cubrimiento de vértices construido V' es a lo sumo el doble del óptimo.

Dada una aproximación polinomial, el *error relativo* ε de las soluciones *y* con respecto a las instancias de problema x se define por:

$$\varepsilon = |opt(x) - m(x, y)| / opt(x).$$

Por ejemplo, considerando el ejemplo anterior:

$$\varepsilon = (m(x, y) - opt(x)) / opt(x) \leq (2opt(x) - opt(x)) / opt(x) = 1.$$

Dicho algoritmo se dice entonces que es *aproximable*, con una ε–*aproximación* tal que $\varepsilon \leq 1$, o lo que es lo mismo, con una 1-aproximación.

Los problemas de NPO aproximables conforman la clase APX. Se prueba que si P $\neq$ NP, entonces existe en NPO un problema no aproximable, es decir, APX $\neq$ NPO.
Otras dos clases, incluidas en APX, son PAS y FPAS. Los problemas de PAS (por *polynomial approximation scheme*) son ε-aproximables para todo ε. Y los problemas de FPAS (por *fully polynomial approximation scheme*) son los problemas de PAS que se resuelven en tiempo polinomial no sólo con respecto a la longitud del input sino también al valor $1/\varepsilon$ (un algoritmo que trabaja en tiempo exponencial con respecto a $1/\varepsilon$, para ε muy pequeño, en la práctica es intratable).
Se cumple que PO $\subseteq$ FPAS $\subseteq$ PAS $\subseteq$ APX $\subseteq$ NPO. Las inclusiones son estrictas a menos que se cumpla P = NP.

Como con los problemas de decisión, también en este caso se utilizan las reducciones y el concepto de completitud para «probar» la pertenencia a las distintas clases de la jerarquía en base a determinadas asunciones. La reducción que se utiliza en este caso se denomina *APX-reducción* o *reducción* $\leq_{APX}$, la cual cumple que si $A \leq_{APX} B$ y $B \in APX$, entonces $A \in APX$. Se define que un problema de optimización A es *NPO-completo*, si $A \in NPO$ y todo problema de optimización B tal que $B \in NPO$ cumple que $B \leq_{APX} A$. Se cumple que si A es NPO-completo entonces no pertenece a APX, a menos que P = NP.
Un ejemplo de problema NPO-completo es minSAT, que consiste en, dada una fórmula booleana ϑ con costos (números naturales) asociados a las variables, encontrar una asignación que satisfaga ϑ tal que la suma de los costos de las variables con valor verdadero sea mínima.
Se conocen pocos problemas NPO-completos en comparación con el tamaño de la clase NPC. La NPO-completitud es más difícil de probar que la NP-completitud, y además, muchos problemas de APX tienen lenguajes asociados NP-completos.

A2.1.2. Problemas de enumeración de #P

Entre los problemas de enumeración, la clase que se destaca es #P (o *sharp P*). Si R es una relación binaria decidible en tiempo determinístico polinomial, el problema de enumeración asociado a R de la forma: «Dado x, ¿cuántos *y* hay tales que R(x, y)?», pertenece a #P.
Por ejemplo, el problema #SAT: «Dada una fórmula booleana, calcular la cantidad de asignaciones de valores de verdad que la satisfacen», pertenece a #P. Lo mismo ocurre con el problema #HC: «Calcular la cantidad de circuitos de Hamilton de un grafo», el problema #CLIQUE: «Calcular la cantidad de cliques de tamaño mayor o igual que K de un grafo», el problema #GRAPH RELIABILITY: «Calcular la cantidad de subgrafos de un grafo tales que contengan un camino del vértice v_1 al vértice v_n», etc.
Se prueba fácilmente que $\#P \subseteq PSPACE$. La idea es que enumerar una a una, en un determinado orden, todas las soluciones de un problema de #P, reutilizando espacio y manteniendo actualizado un contador representado en una base adecuada, consume espacio determinístico polinomial.
En este marco de problemas se definen las *reducciones parsimoniosas* (*parsimonious reductions*), que van de un problema de enumeración

A a un problema de enumeración B y tienen la propiedad de preservar la cantidad de soluciones. Así se definen problemas *#P-completos*, con el significado habitual. Por ejemplo, se prueba que #SAT es #P-completo. La demostración se basa en la utilizada en el Teorema de Cook para la prueba de la NP-completitud del problema de decisión SAT. Un caso muy interesante de problema #P-completo es el de la permanente de una matriz, porque la permanente se define de una manera muy similar al determinante, pero a diferencia de éste, la permanente es muy difícil de calcular en el sentido de la complejidad temporal.
Probar que un problema de enumeración es #P-completo en la práctica es demostrar que es intratable: si se lo resolviera en tiempo determinístico polinomial valdría P = NP.

A2.2. Máquinas de Turing con oráculo

Las MT con oráculo modelizan las MT con «subrutinas». En lo que sigue se describen resumidamente algunas aplicaciones de estas máquinas alternativas, con foco en la complejidad computacional.

Definición. Sea un lenguaje $A \subseteq \Sigma^*$. M es una *máquina de Turing con oráculo A* si M es una MT que:

- Incluye dos cintas especiales, la *cinta de pregunta* y la *cinta de respuesta*.
- Incluye un estado especial, el *estado de pregunta*.
- Trabaja de la forma habitual, pero toda vez que su estado es el de pregunta, según sea el contenido w de la cinta de pregunta, en el siguiente paso la cinta de respuesta tendrá un «1» si $w \varepsilon A$, o bien un «0» si $w \notin A$.

□

Una MT M con oráculo A se denota con M^A, y el estado de pregunta con $q_?$. L es un *lenguaje recursivamente numerable con respecto al oráculo A* si es reconocido por una MT M^A. Si M^A se detiene a partir de todos los inputs, $L = L(M^A)$ es un *lenguaje recursivo con respecto al oráculo A*. Dos oráculos son *equivalentes* si cada uno es recursivo con respecto al otro.

Ejemplo. La siguiente MT M^{HP} reconoce HP y se detiene siempre: dado el input w, copia w en la cinta de pregunta, pasa al estado $q_?$, y acepta o rechaza según en el paso siguiente la cinta de respuesta tenga un «1» o un «0», respectivamente.

El ejemplo anterior muestra que si no se restringe el tipo de oráculo utilizado, se pueden contradecir aserciones (sobre computabilidad, decidibilidad y complejidad) probadas utilizando MT sin oráculo.

A2.2.1. Jerarquías de dificultad de problemas

Por medio de las MT con oráculo se pueden establecer jerarquías de dificultad, no sólo en el conjunto de lenguajes decidibles R, sino también sobre los lenguajes indecidibles.

Por ejemplo, sean los oráculos:

$A_1 = \{<M> \mid L(M) = \varnothing\}$ y $A_2 = \{<M> \mid L(M^{A1}) = \varnothing\}$.

Se prueba que $L_U = \{(<M>, w) \mid M \text{ acepta } w\}$ es recursivo con respecto a A_1. Más aún, es equivalente a A_1. Por otro lado, se prueba que $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\}$ es recursivo con respecto a A_2 y que A_2 es recursivo con respecto a L_{Σ^*}, es decir, L_{Σ^*} y A_2 son equivalentes. Esto indica de alguna manera que L_{Σ^*} es «más indecidible» que L_U y que $L_\varnothing$, con $L_\varnothing = A_1$.
Si bien la diferencia anterior en la práctica no tiene mucha importancia, dado que todos los problemas son indecidibles, cuando se consideran versiones más restringidas de los mismos la situación cambia.
Por ejemplo, para gramáticas G libres de contexto, los problemas «¿w ε L(G)?» y «¿L(G) = $\varnothing$?» son decidibles, mientras que el problema «¿L(G) = Σ^*?» es indecidible. Y en el caso de las gramáticas regulares, en que todos los problemas son decidibles, la diferencia se da entre una solución tratable para los dos primeros casos y una intratable para el último.

A2.2.2. Turing-reducciones

Definición. Sean L_1 y L_2 dos lenguajes incluidos en Σ^*. Existe una *Turing-reducción* de L_1 a L_2 sii existe una MTD M^{L2} que reconoce L_1 y se detiene

siempre, es decir, L_1 es recursivo con respecto a L_2. En particular, si M^{L2} trabaja en tiempo polinomial, la Turing-reducción es *polinomial*.

□

La expresión $L_1 \leq^T L_2$ denota que L_1 es *Turing-reducible* a L_2. Las Turing-reducciones polinomiales de L_1 a L_2 se denotan con $L_1 \leq^T_p L_2$.

Ejemplo. Para todo lenguaje L vale: $L \leq^T L^C$. La siguiente MT M con oráculo L^C reconoce L y se detiene siempre:

1) Copiar el input w en la cinta de pregunta y pasar al estado q_2.
2) Si en el paso siguiente la cinta de respuesta tiene un «1», rechazar, y si tiene un «0», aceptar.

Vale además $L \leq^T_p L^C$: la MT con oráculo L^C reconoce L en tiempo determinístico lineal.

Supóngase en el ejemplo anterior que $L = HP^C$, y por lo tanto: $HP^C \leq^T HP$. Como $HP^C \notin RE$ y $HP \in RE$, a diferencia entonces de lo que ocurre con las reducciones α utilizadas en los capítulos precedentes (para distinguirlas en este contexto se las denominará *reducciones many-one* como también se las conoce), en que si $L_1 \alpha L_2$ y $L_2 \in RE$ se cumple $L_1 \in RE$, ahora con las Turing-reducciones esta propiedad deja de cumplirse.

En cambio, las Turing-reducciones siguen sirviendo para probar la pertenencia a la clase R:

Teorema. Dados L_1, L_2 incluidos en Σ^*, si $L_1 \leq^T L_2$ y $L_2 \in R$, entonces $L_1 \in R$.

Prueba. Si $L_1 \leq^T L_2$, entonces existe una MT M^{L2} que acepta L_1 y se detiene siempre. Y si $L_2 \in R$, entonces existe una MT M_2 que acepta L_2 y se detiene siempre.

La siguiente MT M_1 reconoce L_1 y se detiene siempre: M_1 simula M^{L2} salvo cuando M^{L2} acude al oráculo L_2. En lugar de esto, M_1 simula M_2 a partir del contenido de la cinta que representa la cinta de pregunta de M^{L2}, y escribe un «1» ó un «0» en la cinta que representa la cinta de respuesta de M^{L2} según M_2 acepta o rechaza, respectivamente.

Corolario. Si $L_1 \leq^T L_2$ y $L_1 \notin R$, entonces $L_2 \notin R$.

□

Se cumple que si L_1 es many-one reducible a L_2, también es Turing-reducible a L_2, mientras que la inversa no tiene por qué valer. Por lo tanto, las Turing-reducciones pueden ser en algunos casos más útiles que las reducciones many-one para probar que un lenguaje es o no decidible, cuando no exista una reducción many-one que sirva para la prueba, o cuando exista pero sea muy difícil de construir.

Ejercicio. *Probar que si L_1 es many-one reducible a L_2 entonces L_1 es Turing-reducible a L_2, y que existen L_1 y L_2 tales que L_1 es Turing-reducible a L_2 y L_1 no es many-one reducible a L_2.*

□

Las Turing-reducciones también son alternativa para probar la pertenencia a la clase P, aún cuando no puedan encontrarse reducciones many-one:

Teorema. Dados L_1, L_2 incluidos en Σ^*, si $L_1 \leq^T_p L_2$ y $L_2 \in P$, entonces $L_1 \in P$.

Prueba. La prueba es la misma que la del teorema anterior, pero teniendo en cuenta el tiempo en que trabajan las MT involucradas:
- Como $L_1 \leq^T_p L_2$, entonces la MTD M^{L2} que acepta L_1 trabaja en tiempo polinomial.
- Como $L_2 \in P$, entonces la MT M_2 que acepta L_2 trabaja en tiempo polinomial.
- Por lo tanto, la simulación ya descripta de M^{L2} tarda tiempo polinomial: el tiempo incrementado no altera la cota polinomial, porque existe un número polinomial de preguntas al oráculo para simular, y los contenidos de la cinta que representa la cinta de pregunta miden tamaño polinomial con respecto al input.

Corolario. Si $L_1 \leq^T_p L_2$ y $L_1 \notin P$, entonces $L_2 \notin P$.

□

A2.2.3. Relativización

Las conjeturas $P \neq NP$ y $P = NP$ se pueden *relativizar* utilizando oráculos. Siendo P^A y NP^A las clases de lenguajes reconocibles por MTD y MTN,

respectivamente, con oráculo A, que trabajan en tiempo polinomial, la idea es intentar resolver las conjeturas *no relativizadas* a través de refinamientos sucesivos de resultados obtenidos con oráculos (por ejemplo, a partir de $P^X \neq NP^X$ para determinados oráculos X), hasta llegar a oráculos triviales que permitan hacer referencia a las conjeturas originales. Esto además es extensible a otras conjeturas.
Se prueba que existen oráculos A y B tales que $P^A = NP^A$ y $P^B \neq NP^B$ (lo primero se demostrará enseguida). Por lo tanto, las únicas técnicas capaces de resolver las conjeturas originales deben ser *no relativizables*, es decir, deben tener la propiedad de que lo que prueben sobre conjuntos de lenguajes sin valerse de oráculos, no necesariamente deba valer utilizando oráculos.
Por ejemplo, si se probara por diagonalización que $P \neq NP$, la misma demostración aplicaría utilizando oráculos, pero como existe un oráculo A tal que $P^A = NP^A$, entonces la diagonalización no es un camino válido para demostrar la conjetura. Lo mismo ocurre con la técnica de simulación para probar P = NP, por existir un oráculo B tal que $P^B \neq NP^B$.
En cambio, una posibilidad podría ser basarse en propiedades de clausura, apuntando a demostrar $P \neq NP$, dado que podría pasar que P fuera cerrada y NP no lo fuera con respecto a una operación determinada, lo que no tendría por qué valer con oráculos.
A continuación se prueba uno de los dos casos mencionados previamente:

Teorema. Existe un oráculo A tal que $P^A = NP^A$.

Intuitivamente, la idea es utilizar un oráculo «difícil», de manera tal que las clases P y NP, valiéndose del mismo, resulten iguales.

Prueba. $P^A \subseteq NP^A$ por definición. Para probar $NP^A \subseteq P^A$ se hará A = QBF (el lenguaje de las fórmulas booleanas verdaderas con cuantificadores y sin variables libres) y se demostrará:
a) $NP^A \subseteq$ PSPACE, y (b) PSPACE $\subseteq P^A$. En realidad, la prueba podría utilizar cualquier lenguaje PSPACE-completo.

Parte (a). $NP^A \subseteq$ PSPACE:

Sea L ε NP^A. Sea M_1^A una MTN que reconoce L en tiempo polinomial. Por lo tanto, dado el input w, M_1^A efectúa en cada computación un número polinomial de preguntas a A, siendo toda vez el tamaño del contenido de la cinta de pregunta un polinomio con respecto a la longitud de w.

Sea la siguiente MTD M_2 (que pretende simular determiníticamente M_1^A): toda pregunta al oráculo A se reemplaza ejecutando la MTD que reconoce A en espacio polinomial, y escribiendo luego «1» o «0» adecuadamente.
Se comprueba fácilmente que la MTD M_2 reconoce L y que trabaja en espacio polinomial. Por lo tanto L ε PSPACE, y así: $NP^A \subseteq$ PSPACE.

Parte (b). PSPACE $\subseteq P^A$:

Sea L ε PSPACE. Sea M_2 una MTD que reconoce L en espacio polinomial. Y sea M_f una MTD que reduce en tiempo polinomial el lenguaje L al lenguaje A (existe porque A es PSPACE-completo con respecto a las reducciones poly-time).
Sea la siguiente MTD M_3^A (que pretende simular M_2): dado el input w, ejecuta M_f para calcular f(w) en la cinta de pregunta, luego pasa al estado $q_?$ y pregunta al oráculo A, y finalmente acepta o rechaza según A responda «1» o «0», respectivamente.
Se comprueba fácilmente que la MTD M_3^A reconoce L y que trabaja en tiempo polinomial. Por lo tanto L ε P^A, y así: PSPACE $\subseteq P^A$.

□

A pesar de las implicancias negativas de la relativización de las conjeturas P ≠ NP y P = NP, hay también *relativizaciones positivas*, en el sentido de que se pueden establecer relaciones útiles entre clases de complejidad con y sin oráculos.

Por ejemplo, sea NP_b^A la clase de los lenguajes reconocibles por MTN con oráculo A tales que hacen a lo sumo un número polinomial de preguntas. Se prueba que P = NP sii para todo oráculo A vale que $P^A = NP_b^A$.

Otro ejemplo se basa en limitar ya no la cantidad de preguntas a los oráculos sino la potencia computacional de los mismos. Por ejemplo, se prueba que P = NP sii para todo lenguaje tally T vale que $P^T = NP^T$. De este modo, dos caminos válidos para probar la conjetura P ≠ NP son encontrar un oráculo A tal que $P^A \neq NP_b^A$ o bien un lenguaje tally T tal que $P^T \neq NP^T$.

La *Jerarquía Polinomial* es otro caso de relativización positiva:
Si $P^C = \bigcup_{L \varepsilon C} P^L$ y $NP^C = \bigcup_{L \varepsilon C} NP^L$, la Jerarquía Polinomial es un conjunto infinito de clases de lenguajes $\Delta_k P$, $\Sigma_k P$, $\Pi_k P$, que se definen de la siguiente manera:

- $\Delta_0 P = P$, $\Delta_1 P = P$, y en general $\Delta_{k+1} P = P^{\Sigma_k P}$ para $k \geq 1$.
- $\Sigma_0 P = P$, $\Sigma_1 P = NP$, y en general $\Sigma_{k+1} P = NP^{\Sigma_k P}$ para $k \geq 1$.
- $\Pi_0 P = P$, $\Pi_1 P = CO\text{-}NP$, y en general $\Pi_{k+1} P = CO\text{-}NP^{\Sigma_k P}$ para $k \geq 1$.

Por ejemplo, las primeras clases de la jerarquía son P, NP, CO-NP (primer nivel), P^{NP}, NP^{NP}, $CO\text{-}NP^{NP}$ (segundo nivel), etc.

La unión infinita de los $\Sigma_k P$ se denota con PH. Se prueba que PH $\subseteq$ PSPACE. Así, la Jerarquía Polinomial agrupa clases de problemas que van desde P hasta PSPACE.

El siguiente ejemplo muestra un problema conocido de la clase $\Sigma_2 P$:

Ejemplo. Sea EF = {(ϑ, K) | ϑ es una fórmula booleana, K es un número natural, y se cumple que existe una fórmula booleana ϑ' equivalente a ϑ tal que tiene a lo sumo K literales}.
Se prueba que EF pertenece a la clase $\Sigma_2 P = NP^{NP}$, utilizando por ejemplo el oráculo SAT.

No se sabe si a partir de un número dado k, vale $\Sigma_k P = \Sigma_{k+1} P$, lo que se denomina *colapso* de la Jerarquía Polinomial. Se prueba que $P \neq NP$ sii $P \neq PH$, y que aún asumiendo $P \neq NP$ la Jerarquía Polinomial podría igual colapsar. Asumiendo que no hay colapso, encontrar un lenguaje L que sea $\Sigma_k P$-completo significa que se identifica el nivel exacto en que se encuentra L en la jerarquía:

Ejemplo. Sea k-QBF = {ϑ | ϑ es una fórmula booleana verdadera con cuantificadores, de la forma $(\exists x_1)(\forall x_2)(\exists x_3)(\forall x_4)\ldots(Qx_k)[E(x_1, x_2, x_3, x_4, \ldots, x_k)]$, con Q = $\exists$ ó $\forall$ según k sea impar o par}.

Se prueba que k-QBF es $\Sigma_k P$-completo con respecto a las reducciones poly-time.

Por el ejemplo, se cumple que el lenguaje 2-QBF es completo en la clase NP^{NP} con respecto a las reducciones poly-time. En cuanto al

lenguaje EF, si bien está en NP^{NP} no se sabe si es completo en dicha clase.
Se han encontrado pocos problemas completos en la Jerarquía Polinomial. En particular, no se conocen problemas PH-completos. Se prueba que si existe un problema PH-completo, entonces la jerarquía colapsa en algún nivel. Por lo tanto, como PSPACE tiene problemas completos, si valiese la igualdad PH = PSPACE la Jerarquía Polinomial colapsaría.

A2.3. Máquinas de Turing probabilísticas

Se ha hecho referencia previamente a los tiempos promedio de los algoritmos, en que deben considerarse las distribuciones probabilísticas de la población de los inputs, lo que comúnmente es una tarea bastante difícil. Cabe destacar que existen problemas NP-completos que resultan tratables si se considera el tiempo promedio.
Otra aplicación de la teoría de probabilidades, en el marco de la complejidad computacional, es la construcción de algoritmos probabilísticos. Se trata de algoritmos determinísticos de tiempo polinomial pero con elecciones aleatorias en algunos pasos, lo que puede producir diferentes outputs en diferentes ejecuciones. Conociendo la distribución de los outputs, se efectúan ciertas asunciones con determinada probabilidad.
El modelo computacional asociado a los algoritmos probabilísticos lo constituyen las *máquinas de Turing probabilísticas*. Las MT probabilísticas (o PT) son MT no determinísticas que trabajan en tiempo polinomial y tienen un criterio de aceptación distinto al de las MTN estudiadas hasta el momento: una PT M acepta o rechaza un input w según la cantidad de computaciones que terminan en el estado q_A, o en otras palabras, según la probabilidad de que una computación de M acepte.
Dada una PT M y un input w, la expresión $\alpha(M, w)$ denota la razón entre la cantidad de computaciones de M que aceptan w y la cantidad total de computaciones de M, y simétricamente, la expresión $\beta(M, w)$ denota la razón entre la cantidad de computaciones de M que rechazan w y la cantidad total de computaciones de M. Si M acepta w, la *probabilidad de error* de M es $\beta(M, w)$, y si rechaza, es $\alpha(M, w)$.
Se definen distintas clases de PT, según el criterio de aceptación que se adopte.

Por ejemplo, las *máquinas R* (*R-machines*) son PT M tales que:

- Para todo w, $\alpha(M, w) > 1/2$ ó $\beta(M, w) = 1$.
- M acepta w, si $\alpha(M, w) > 1/2$.
- M rechaza w, si $\beta(M, w) = 1$.

Ejemplo. La siguiente PT M (máquina R) determina si un número N es compuesto. El algoritmo se basa en un teorema de Fermat, que establece que si un número $N > 2$ es primo, entonces para todo número natural *a* tal que $1 \leq a < N$, se cumple que $a^{N-1} = 1 \pmod N$. Así, encontrando un número *a* entre 1 y N – 1 tal que $a^{N-1} \neq 1 \pmod N$, se prueba que N es compuesto. El número *a* se denomina testigo, y se demuestra que si N es un número compuesto impar, admite al menos (N – 1) / 2 testigos.

Dado el input N, la PT M hace:

1) Si N es par, aceptar.
2) Hacer a := random (1, N – 1).
3) Si a es testigo, aceptar, y en caso contrario, rechazar.

La función random (1, N – 1) selecciona aleatoriamente un número entre 1 y N – 1. Se prueba que determinar si *a* es testigo tarda tiempo determinístico polinomial. Notar que si N es primo, se cumple que $\beta(M, N) = 1$ (todas las computaciones de M rechazan) y así la probabilidad de error es cero, y si N es compuesto, se cumple que $\alpha(M, w) > 1/2$ (más de la mitad de las computaciones de M aceptan) y así la probabilidad de error es menor que 1/2. Por lo tanto, M decide el problema de composicionalidad.

Siguiendo con el ejemplo, la probabilidad de error para determinar si un número es compuesto de manera probabilística y en tiempo determinístico polinomial se puede hacer muy pequeña de la siguiente manera:

- Se itera K veces la PT M a partir de un mismo número N, por ejemplo con K = 100.
- Si en alguna iteración M acepta, entonces se acepta: N es compuesto porque es par o porque es impar y tiene un testigo. En este caso la probabilidad de error es cero.

– Si en cambio en las 100 iteraciones M rechaza, entonces se rechaza: N es primo (y por eso no se encuentran testigos), o bien N es compuesto. En este último caso la probabilidad de error es menor que 2^{-100} (toda iteración debe considerarse independiente de las otras).

Este mecanismo aplica a cualquier problema decidible por una máquina R. La clase de complejidad computacional que agrupa a los lenguajes decidibles por máquinas R se denomina R. Las máquinas R nunca aceptan mal. Pueden rechazar mal, pero con una probabilidad muy pequeña si se itera un número polinomial de veces con respecto a la longitud de los inputs, de modo tal de seguir trabajando en tiempo eficiente.

Otros tipos de PT son:

– Las *máquinas PP* (*PP-machines*). Dados la PT M y el input w, M acepta w si $\alpha(M, w) > 1/2$, o rechaza w si $\beta(M, w) \geq 1/2$.
– Las *máquinas BPP* (*BPP-machines*). Dados M, w y una constante ε de (0, 1/2), se cumple para todo w que $\alpha(M, w) > 1/2 + \varepsilon$, ó $\beta(M, w) > 1/2 + \varepsilon$, y M acepta w si $\alpha(M, w) > 1/2 + \varepsilon$, o rechaza w si $\beta(M, w) > 1/2 + \varepsilon$.
– Las *máquinas ZPP* (*ZPP-machines*). Dados M y w, se cumple para todo w que $\alpha(M, w) > 1/2$ y $\beta(M, w) = 0$ ó $\beta(M, w) > 1/2$ y $\alpha(M, w) = 0$, y M acepta w si $\alpha(M, w) > 1/2$, o rechaza w si $\beta(M, w) > 1/2$. Estas máquinas tienen un estado final extra, el estado *don't know*, que las hace más confiables que las otras, porque cuando no hay certeza responden «no sé». Los algoritmos de este tipo se conocen como *algoritmos Las Vegas*, mientras que los anteriores se denominan *algoritmos Monte Carlo*.

Las máquinas PP son las menos útiles en la práctica, dado que para llegar a obtener una probabilidad de error arbitrariamente pequeña, puede ser que deban ejecutarse un número exponencial de veces a partir de un mismo input. Las máquinas BPP y ZPP en cambio, al igual que las máquinas R, logran probabilidades de error muy bajas iterándose a lo sumo una cantidad polinomial de veces.
Las clases de problemas asociadas a las máquinas mencionadas son, respectivamente, PP, BPP y ZPP. Se cumple:

$$P \subseteq ZPP \subseteq R \subseteq NP \subseteq PP \subseteq PSPACE, \text{ y } R \subseteq BPP \subseteq PP.$$

No se sabe si las inclusiones son estrictas.
No se conocen lenguajes completos con respecto a las reducciones poly-time en las clases ZPP, R y BPP. En cambio, por ejemplo el lenguaje maxSAT = {(ϑ, K) | ϑ es una fórmula booleana satisfactible por más de K asignaciones de valores de verdad} se prueba que es PP-completo con respecto a las reducciones poly-time.
Las clases ZPP, R y BPP deberían considerarse conjuntos de problemas tratables. De esta manera, no debería suceder que alguna máquina ZPP, R o BPP resuelva algún problema NP-completo. Una aplicación interesante de estas máquinas sería resolver más simple o eficientemente que las MT determinados problemas de P y de NPI.

A2.4. Máquinas para paralelismo

Existen varios modelos para las computadoras paralelas. Desde el punto de vista de la complejidad computacional, el objetivo principal de la modelización es caracterizar a los problemas cuyas soluciones mediante el procesamiento paralelo son significativamente más rápidas que sus soluciones secuenciales.
Los dos modelos más utilizados son las *familias uniformes de circuitos booleanos* y las *máquinas PRAM* (por *Parallel Random Access Machines*), que son la adaptación al paralelismo de las máquinas RAM vistas en la primera parte del libro.

Una familia uniforme C de circuitos booleanos es una secuencia C = {c_n | n $\geq$ 1}, tal que cada c_n es un circuito booleano acíclico con n nodos de input y un nodo de output (en la visión de c_n como circuito calculador de una función se deben considerar m nodos de output). c_n acepta un input w si el output producido es 1 (que representa el valor verdadero). La familia C acepta la unión de los lenguajes reconocidos por los circuitos c_n que la componen.
El *tamaño* de c_n es la cantidad de sus nodos intermedios o puertas (*gates*), y se denota con H(c_n). La *profundidad* de c_n es la longitud del camino más largo desde uno de sus nodos de input a su nodo de output, y se denota con T(c_n). El tamaño de C, que representa la cantidad de procesadores, es la función $H_C(n) = H(c_n)$. La profundidad de C, que representa el tiempo paralelo, es la función $T_C(n) = T(c_n)$.
El *costo* de C es el producto de su tamaño y su profundidad, o en otras palabras, es la cantidad de pasos ejecutados en conjunto por los procesadores. Como todo algoritmo paralelo puede simularse por

un algoritmo secuencial, este valor no puede ser menor que el mejor tiempo de una solución secuencial equivalente.
La familia C se dice uniforme porque se establece la condición de que exista una MTD que compute la función $1^n \rightarrow c_n$ en espacio $O(\log(H(c_n)))$. Intuitivamente, la idea es que todos los circuitos representan un mismo algoritmo. Sin una restricción de este tipo podrían producirse resultados inconsistentes, tales como que familias de circuitos muy simples reconozcan lenguajes muy difíciles. Se prueba que un lenguaje $L \varepsilon P$ sii existe una familia uniforme de circuitos booleanos de tamaño polinomial que lo reconoce, por lo que este modelo también podría contribuir a probar la conjetura $P \neq NP$.

Las máquinas PRAM son otro modelo muy utilizado en el marco del procesamiento paralelo. Son máquinas sincrónicas, en el sentido de que todos los procesadores ejecutan una instrucción al mismo tiempo. Todas las instrucciones tardan lo mismo. Existe una única memoria global compartida por los procesadores.
Hay distintas variedades de máquinas PRAM, según cómo resuelvan los conflictos de lectura y escritura, y qué tipo de instrucciones puedan ejecutar al mismo tiempo en los procesadores.
También en este modelo se establece una condición de uniformidad: debe existir una MTD que genere un algoritmo PRAM en espacio logarítmico con respecto a la longitud de los inputs. Es decir, las distintas máquinas PRAM, según el tamaño de los inputs, deben constituir todas un mismo algoritmo.
Más allá de las ventajas de un modelo sobre el otro, los dos satisfacen una relación que permite utilizarlos indistintamente asegurando resultados consistentes. Esta relación se llama Tesis de la Computación Paralela, y establece que, dados dos polinomios Q_1 y Q_2 y el modelo $\mathbb{P}$:

$$\mathbb{P}\text{-TIME}(T(n)) \subseteq \text{DSPACE}(Q_1(T(n))) \text{ y } \text{DSPACE}(S(n)) \subseteq \mathbb{P}\text{-TIME}(Q_2(S(n))),$$

donde $\mathbb{P}$-TIME(T(n)) es la clase de lenguajes reconocidos por el modelo $\mathbb{P}$ (familia de circuitos booleanos o máquinas PRAM) en tiempo O(T(n)). En otras palabras, la tesis establece que el tiempo paralelo se relaciona polinomialmente con el espacio determinístico secuencial.

El siguiente ejemplo presenta dos algoritmos paralelos expresados con máquinas PRAM:

Ejemplo. La siguiente máquina PRAM M suma N números en paralelo. Para facilitar la presentación, se asume que N es una potencia de 2, y se utiliza una notación tipo Pascal, en la que se agrega la sentencia «for i := 1 to k do in parallel <instrucción>», que se interpreta de la siguiente manera: se activan los primeros k procesadores de la máquina, y éstos ejecutan en paralelo <instrucción>, por lo que la sentencia tarda lo que tarda <instrucción>.

Dado el input {N, x_1, x_2, x_3, …, x_N}, M hace:

for i := 1 to log(N) do
for k := 1 to $N/2^i$ do in parallel $x_k := x_{2k-1} + x_{2k}$

Por ejemplo, si N = 8, la PRAM M primero suma en paralelo x_1 con x_2, x_3 con x_4, x_5 con x_6, y x_7 con x_8. Después suma en paralelo los dos pares de números obtenidos. Y finalmente suma los últimos dos resultados. Se comprueba fácilmente que M utiliza N/2 procesadores, y que trabaja en tiempo logarítmico.

Por otro lado, la siguiente máquina PRAM M ordena en paralelo, en orden creciente, N números. La idea es que la posición del número x_k se calcula sumando la cantidad de números x_i menores o iguales que x_k. Se utiliza la suma paralela de números del ejemplo anterior:

Dado el input {x_1, x_2, x_3, …,x_N}, M hace:

for i, k := 1 to N do in parallel
if $x_i \leq x_k$ then a[i, k] := 1 else a[i, k] := 0
for k := 1 to N do in parallel
b[k] := $\Sigma_{i = 1, N}$ a[i, k], utilizando la suma paralela del ejemplo anterior
for i := 1 to N do in parallel
y[b[i]] := x_i

El vector *y* tiene los números x_i en orden creciente. Notar que si un valor x_i se repite k veces, entonces aparece una sola vez en y, estando precedido por (k – 1) componentes vacías. La modificación del algoritmo para permitir componentes iguales en el output es muy sencilla. Se comprueba fácilmente que M utiliza N^2 procesadores, y que trabaja en tiempo logarítmico.

La clase de lenguajes que pueden reconocerse por familias uniformes de circuitos booleanos de tamaño polinomial y profundidad polilogarítmica con respecto al tamaño de los inputs se denomina NC. NC caracteriza a los problemas de decisión con solución paralela eficiente utilizando este modelo. FNC se define de manera similar para el caso de las funciones.
Que el tamaño de las familias de circuitos booleanos sea polinomial, no sólo apunta a la similitud con las computadoras, sino que también asegura que la modelización no sea demasiado potente con respecto a las máquinas de Turing, como se verá enseguida en un ejemplo.
NC es la unión infinita de clases NC^k, con $k \geq 1$. Se cumple que $L \varepsilon NC^k$ si existe una familia uniforme C de circuitos booleanos y una constante h tales que:

- C acepta L.
- $H_C(n) = O(n^h)$.
- $T_C(n) = O(\log^k n)$.

Como en otras jerarquías vistas antes, existen problemas conocidos en los primeros niveles de NC. Por ejemplo, las cuatro operaciones aritméticas pertenecen a NC^1.
Por su parte, en el marco de las máquinas PRAM, la clase de problemas que pueden resolverse con una cantidad polinomial de procesadores en tiempo polilogarítmico con respecto al tamaño de los inputs se denomina AC, y es la que caracteriza a los problemas con solución paralela eficiente. AC es la unión infinita de clases AC^k, con $k \geq 1$, y se prueba que AC = NC.
El siguiente ejemplo justifica por qué es necesario acotar a un número polinomial, la cantidad de procesadores del modelo paralelo:

Ejemplo. Con 2^n procesadores se puede decidir en tiempo paralelo polinomial si un grafo G tiene un cubrimiento de vértices de tamaño a lo sumo K (problema que se sabe es NP-completo). Dados G y K, sea la siguiente máquina PRAM M:

1) Cada procesador P_i analiza su propio índice i (número binario). Si la cantidad de «1» en i es a lo sumo K, y los vértices de G asociados a los «1» forman un cubrimiento de G, entonces P_i acepta el input.
2) Si algún procesador aceptó el input, entonces M acepta, y en caso contrario, rechaza.

Es fácil comprobar que cada procesador trabaja en tiempo polinomial con respecto a n = |(G, K)|, y por lo tanto M trabaja en tiempo paralelo polinomial.

En realidad, con una cantidad exponencial de procesadores podría resolverse en tiempo paralelo polinomial cualquier problema NP-completo.

Algunos ejemplos de problemas con solución paralela eficiente son los siguientes:

- La suma y el ordenamiento de n números tardan O(log(n)) pasos, respectivamente con n/2 y n^2 procesadores.
- El producto de dos matrices cuadradas A y B de n^2 elementos tarda O(log(n)) pasos, con n^3 procesadores.
- La búsqueda del camino mínimo en un grafo de n vértices con arcos con costo tarda $O(\log^2 n)$ pasos, con n^2 procesadores.
- La potencia A^n dada una matriz cuadrada A de n^2 elementos tarda $O(\log^2 n)$ pasos, con $n^3/2$ procesadores.

Se prueba que $NC \subseteq P$. La demostración se basa en que:

- Los circuitos c_n pueden ser generados en espacio determinístico $O(\log(H(c_n)))$ y así en tiempo determinístico polinomial con respecto a $H(c_n)$), por la condición de uniformidad.
- El problema de evaluación de circuitos o CIRCUIT VALUE ya referido anteriormente: «Dado el circuito c y el input w, ¿acaso se cumple que c(w) = verdadero?», pertenece a P.

No se sabe si P incluye estrictamente a NC. Como se cumple que la clase NC es cerrada con respecto a las reducciones log-space, probar que un lenguaje L es P-completo con respecto a las reducciones log-space, en la práctica implica probar que $L \notin NC$ (a menos que P = NC). Ya se han mostrado anteriormente problemas P-completos, entre los que se encuentra el problema de evaluación de circuitos.
También se prueba que $NC^1 \subseteq DLOGSPACE \subseteq NLOGSPACE \subseteq NC^2$.
Por lo tanto, si $NC^1 \subset DLOGSPACE$, entonces encontrar un lenguaje L que sea DLOGSPACE-completo con respecto a un tipo de reducción adecuada, en la práctica implicará demostrar que L no admite una solución paralela de tiempo O(log(n)).
Tal como ocurre en NP, en que asumiendo $P \neq NP$ se prueba que existe la clase NPI, asumiendo $NC \neq P$ podría darse que en P existieran problemas no P-completos fuera de NC.

Notar que asumiendo las conjeturas NC $\subset$ P y DLOGSPACE $\subset$ P, probar que un lenguaje L es P-completo significa que no admite una solución eficiente en términos de tiempo paralelo ni de espacio determinístico secuencial. En otras palabras, toda solución para L requeriría mucha información simultánea en memoria. Enseguida se verá otra aplicación de la P-completitud.
Para aprovechar la potencia de los algoritmos probabilísticos con error acotado, se utilizan también *modelos paralelos probabilísticos*.
Por ejemplo, un *circuito booleano probabilístico* c_n incluye un número polinomial de inputs aleatorios $y_1, y_2, \ldots, y_{p(n)}$, siendo p(n) algún polinomio, y se define que la probabilidad de que c_n produzca el output 1 es la fracción de las combinaciones de inputs aleatorios y_i tales que c_n produzca 1 a partir de un input w de tamaño n. El criterio de aceptación es que la probabilidad de error sea a lo sumo $1/2 - \varepsilon$, con $0 < \varepsilon \leq 1/2$.
La clase RNC^k, con $k \geq 1$, agrupa a los lenguajes aceptados por familias uniformes de circuitos booleanos probabilísticos de tamaño polinomial y profundidad $O(\log^k(n))$ con respecto a inputs de longitud n. RNC es la unión infinita de las clases RNC^k. De modo similar se define la clase de funciones FRNC.
RNC incluye problemas conocidos, como el problema de determinar si un grafo bipartito tiene un *matching* perfecto. Un grafo G = (V, E) es bipartito si V se divide en V_1 y V_2 y todos los arcos de E tienen un extremo en V_1 y el otro en V_2. Y un *matching* perfecto es un conjunto de arcos E' incluido en E tal que para todos los vértices v de V existe uno y solo un arco e de E' tal que v es extremo de e.
No se conoce la relación entre los conjuntos NC, RNC y P. Dado que no pareciera que RNC = P, entonces probar que un lenguaje L es P-completo con respecto a las reducciones log-space, en la práctica sería probar que L no pertenece a la clase RNC.
También se definen las *máquinas PRAM probabilísticas*, en las que los procesadores tienen la posibilidad de efectuar elecciones aleatorias.

A2.5. Otras máquinas

A2.5.1. Sistema de pruebas interactivas

Un *sistema de pruebas interactivas* (P, V) está formado por dos máquinas de Turing, P (por *probador*) y V (por *verificador*).
Las MT P y V se intercambian mensajes. P le envía mensajes a V a través de la cinta P-a-V que es de solo escritura para P y de solo lectura

para V, y V le envía mensajes a P a través de la cinta V-a-P que es de solo escritura para V y de solo lectura para P. P y V tienen además sus propias cintas de trabajo, y leen un mismo input w que se encuentra en una cinta de solo lectura.
P y V se turnan para estar activas. Cuando una MT está activa envía o recibe mensajes o bien ejecuta operaciones internas. V inicia y termina la computación, aceptando o rechazando, en cuyo caso el sistema (P, V) acepta o rechaza el input w, respectivamente.
La MT P es determinística y no tiene restricciones de espacio ni de tiempo. La cantidad y longitud de los mensajes intercambiados entre P y V es polinomial con respecto a n = |w|.
Si la MT V es determinística y trabaja en tiempo polinomial, el sistema se denomina *sistema de pruebas interactivas determinísticas*, y la clase de lenguajes aceptados es DIP.
Se cumple que DIP = NP. De hecho, este tipo de sistema es una generalización de las MTN que trabajan en tiempo polinomial (que se caracterizaron antes como verificaciones eficientes de pruebas de teoremas). Ahora, en lugar del envío de un solo mensaje de tamaño polinomial desde el probador hacia el verificador (o sea desde el que debe convencer de la correctitud de la prueba hacia el que debe verificar que la prueba es correcta), existe una cantidad polinomial de mensajes de tamaño polinomial que el probador y el verificador se intercambian, no obstante lo cual la clase de lenguajes aceptados es la misma.
En una variante de este sistema, V es una MT probabilística. En este caso, la clase de lenguajes reconocidos pareciera incluir estrictamente a NP. El criterio es el siguiente: V acepta (y por lo tanto el sistema (P, V) acepta el input w) sii $\alpha(P, V, w) > 2/3$ (utilizando la misma notación relacionada con las máquinas de Turing probabilísticas). Se prueba que en lugar del valor 2/3 se puede utilizar cualquier constante mayor que 1/2.
Este tipo de sistema se denomina *sistema de pruebas interactivas*, y la clase de lenguajes aceptados es IP.
El ejemplo siguiente muestra un lenguaje de IP. Para facilitar la presentación, se utiliza una notación tipo Pascal, en la que se agregan las sentencias «transmit» y «receive» para denotar el envío y la recepción de mensajes, respectivamente.

Ejemplo. Sea otra vez el problema de los grafos isomorfos. El lenguaje que lo representa es:

$L = \{(G_1, G_2) \mid G_1 \text{ y } G_2 \text{ son isomorfos}\}$. Se cumple que $L^c \in IP$.

Sea el siguiente sistema de pruebas interactivas (P, V), a partir del input $w = (G_1, G_2)$:

La MT probabilística V hace:

```
exito := true
for k := 1 to 2 do
      i := random (1, 2)
      transmit G_i
      receive m
      if i ≠ m then exito := false
 if exito then aceptar else rechazar
```

y la MT determinística P hace:

```
receive G
if G es isomorfo a G_1 then transmit 1 else transmit 2
```

Se cumple que V trabaja en tiempo polinomial. Hay dos intercambios de mensajes, y el tamaño de los mensajes es polinomial con respecto a la longitud del input $w = (G_1, G_2)$.
Además se cumple que si G_1 y G_2 no son isomorfos, V acepta con probabilidad 1 porque P siempre devuelve el valor m = i. En cambio, si G_1 y G_2 son isomorfos, P devuelve las dos veces el valor m = i sólo cuando V le envía las dos veces el valor 1, por lo que la probabilidad de que V acepte es 1/4.

No se conoce solución no determinística polinomial para el problema anterior, por lo que pareciera entonces que NP $\subset$ IP, lo que se refuerza por el hecho de que se prueba que IP = PSPACE.
Una tercera variante de los sistemas de pruebas interactivas son los *chequeos probabilísticos de pruebas* (*probabilistic checking of proofs*). Están más orientados a la problemática de la criptografía (por ejemplo, al concepto de las pruebas de conocimiento cero o *zero knowledge proofs*). Ahora se asume que el verificador tiene acceso aleatorio a cualquier parte de la prueba, y la cuestión principal es determinar cuánto necesita conocer el verificador de la prueba para convencerse de su correctitud.
La clase de lenguajes asociada a este tipo de sistema se denomina PCP(f,g), tal que si n es el tamaño del input w, la cantidad de bits

aleatorios que se acceden de la prueba es O(f(n)) y la cantidad de accesos es O(g(n)).
Otra aplicación de los chequeos probabilísticos de pruebas se relaciona con las aproximaciones polinomiales de los problemas de optimización: utilizando este tipo de sistema se puede probar que determinados problemas no son aproximables (tal es el caso de maxCLIQUE), aún cuando no se tenga una prueba de que son NPO-completos.

A2.5.2. Máquinas de Turing alternantes

Las *máquinas de Turing alternantes* (*alternating Turing machines*), denotadas con AT, son otra generalización de las máquinas de Turing no determinísticas.
En una MTN M, su árbol de computaciones asociado T_M tiene todos los nodos del tipo «existencial», en el sentido de que el valor booleano del nodo raíz de cualquier subárbol T'_M es verdadero si alguna de sus ramas es de aceptación. En base a esto, inductivamente, se define el valor booleano del nodo raíz del árbol completo T_M.
En cambio, en una AT A, los nodos de su árbol de computaciones asociado T_A son «existenciales» o «universales». Esta diferenciación se debe a que existe una partición del conjunto de estados Q de las AT entre estados existenciales y universales. Ahora, la evaluación del nodo raíz v de cualquier subárbol T'_A se hace de la siguiente manera:

- Si v es existencial, entonces su valor booleano es verdadero sii alguna rama de T'_A es de aceptación.
- Si v es universal, entonces su valor booleano es verdadero sii todas las ramas de T'_A son de aceptación.

En base a esto, inductivamente, se define el valor booleano del nodo raíz del árbol completo T_A.
Una AT A acepta un input w si el valor booleano del nodo raíz del árbol de computaciones T_A asociado a A, considerando w, es verdadero. Así, se determina el lenguaje L = L(A) aceptado por la AT A.
Se definen las siguientes clases de complejidad relacionadas con las AT:

- ATIME(T(n)) es el conjunto de lenguajes aceptados por AT, tales que a partir de w, ninguna computación supera los O(T(|w|) pasos.
- ASPACE(S(n)) es el conjunto de lenguajes aceptados por AT, tales que a partir de w, ninguna computación utiliza más de O(S(|w|) celdas.

En particular, AP = ATIME(n^k) cualquiera sea k y ALOGSPACE = ASPACE(log(n)).

Ejemplo. El lenguaje prenex-QBF = {ϑ | ϑ es una fórmula booleana con cuantificadores en la forma normal prenex (es decir que los cuantificadores aparecen al comienzo de la fórmula), no tiene variables libres y es verdadera} pertenece a AP. El lenguaje prenex-QBF se reconoce con una AT A de la siguiente manera:

1) En cada computación se asignan no determinísticamente valores de verdad a las variables x_1, x_2, x_3, ... Si la variable considerada está cuantificada con $\exists$, la asignación se hace en un estado existencial, y si en cambio está cuantificada con $\forall$, la asignación se hace en un estado universal.
2) Una computación acepta sii la fórmula booleana es verdadera considerando la asignación generada.

Se comprueba fácilmente que L(A) = prenex-QBF. Además, la evaluación de una fórmula booleana considerando una asignación de valores de verdad determinada se resuelve en tiempo determinístico polinomial.

Por otro lado, el lenguaje MONOTONE CIRCUIT VALUE = {(c, w): c es un circuito booleano que no tiene nodos NOT y c(w) = verdadero} pertenece a ALOGSPACE. MONOTONE CIRCUIT VALUE se reconoce con una AT A de la siguiente manera (se asume que un circuito c se representa mediante una lista de nodos y una lista de arcos, como cualquier grafo, pero ahora además los nodos que no son de input tienen asociado un tipo OR o AND, y los nodos de input tienen asociado un valor verdadero o falso):

1) Si el nodo de output m es de tipo AND, A pasa a un estado universal, y si es de tipo OR, A pasa a un estado existencial.
2) Luego A procesa no determinísticamente los predecesores m_i de m (para esto basta con memorizar m y recorrer la lista de arcos). Esto se hace hasta llegar a los nodos de input.
3) Si A alcanza un nodo de input verdadero acepta, y rechaza en caso contrario.

Se comprueba fácilmente que L(A) = MONOTONE CIRCUIT VALUE. Además, como A sólo debe memorizar el nodo que está siendo procesado, cada computación no ocupa más que log(n) celdas, con n = |(c, w)|.

Se prueba que ALOGSPACE = P y que AP = PSPACE. En general, se cumple que el espacio alternante se puede simular en tiempo determinístico con un grado exponencial mayor, y que el tiempo alternante se puede simular en espacio determinístico con el mismo grado exponencial.

La prueba de que ALOGSPACE = P se basa en lo siguiente:
- MONOTONE CIRCUIT VALUE es ALOGSPACE-completo con respecto a las reducciones log-space.
- MONOTONE CIRCUIT VALUE es P-completo con respecto a las reducciones log-space.
- ALOGSPACE y P son cerradas con respecto a las reducciones log-space.

Con el mismo esquema se prueba que AP = PSPACE: el lenguaje prenex-QBF es AP-completo y PSPACE-completo con respecto a las reducciones log-space, y las dos clases son cerradas con respecto a este tipo de reducción.

De esta manera, se han encontrado dos caracterizaciones de la clase PSPACE:
a) IP, en términos de los sistemas interactivos de pruebas.
b) AP, en términos de las MT alternantes.
En este sentido, cabe destacar la estrecha relación que existe, entre otros, entre: los problemas de juegos entre jugadores blancos y negros; los problemas de lenguajes de fórmulas booleanas cuantificadas del tipo prenex-QBF; y los problemas de lenguajes de fórmulas booleanas cuantificadas con cuantificadores aleatorios del tipo $\exists x_1 R x_2 \exists x_3 R x_4 \ldots \vartheta(x_1, x_2, x_3, x_4, \ldots, x_n)$, donde R es un cuantificador aleatorio y así la fórmula booleana es verdadera con una determinada probabilidad (estos lenguajes se asocian por ejemplo a problemas sobre *scheduling* de tareas).

Se puede establecer también una relación entre las AT que trabajan en tiempo polinomial y la Jerarquía Polinomial presentada anteriormente. Se dice que una AT A es *k-alternante* si el máximo número de alternancias entre estados existenciales y universales observado, considerando todas las computaciones de A, es k. Se prueba que para todo $k \geq 0$, un lenguaje L pertenece a la clase $\Sigma_{k+1}P$ (respectivamente, a la clase $\Pi_{k+1}P$) sii existe una AT k-alternante con estado inicial existencial (respectivamente, universal) que reconoce L en tiempo polinomial.

Como en modelos anteriores, también en este caso se pueden definir máquinas probabilísticas.
Las *máquinas de Turing alternantes probabilísticas* tienen la posibilidad de efectuar elecciones aleatorias a lo largo de sus computaciones, y trabajan en tiempo polinomial.
Las clases de lenguajes APP y ABPP son las versiones de las clases PP y BPP mencionadas antes, considerando ahora las AT probabilísticas. En este caso, sin embargo, se cumple que APP = ABPP. Más aún, vale que APP = ABPP = AP = IP = PSPACE.

A2.5.3. Máquinas cuánticas

Las *máquinas cuánticas*, basadas en la mecánica cuántica, al día de hoy no cuentan con implementaciones reales. Los *algoritmos cuánticos* que se ejecutan en ellas pueden introducir mejoras significativas desde el punto de vista de la complejidad temporal de los problemas. De hecho, se han encontrado algoritmos cuánticos que resuelven eficientemente problemas de los que no se conocen soluciones polinomiales utilizando máquinas convencionales.
En lugar del bit, la unidad de información utilizada en las máquinas cuánticas es el *qubit* o *bit cuántico*. Mientras que el valor de un bit es 1 ó 0, el de un qubit es 1 y 0 simultáneamente, con una determinada probabilidad. Es decir, un vector de n qubits puede representar simultáneamente 2^n secuencias de 1 y 0. De este modo, por ejemplo son necesarios 4 pares de bits para representar la misma información que puede representar un vector de 2 qubits; 8 ternas de bits son el equivalente a un vector de 3 qubits; y así sucesivamente. Esta es la base de la potencia computacional de las máquinas cuánticas comparadas con el resto, lo que se conoce como *superposición cuántica*. Se logra paralelismo sin necesidad de aumentar el número de procesadores.
Las computaciones cuánticas se estructuran en tres pasos:

a) Preparación de los qubits en un estado inicial determinado.
b) Transformación de los qubits.
c) Medición del valor de los qubits.

Las computaciones son probabilísticas, cada ejecución puede producir un resultado distinto. El resultado de una computación cuántica es una distribución de probabilidades de los diferentes resultados. Como en el

caso de las MT probabilísticas, la idea es tratar con errores probabilísticos acotados, de modo tal que tiendan a cero iterando computaciones sin que se altere significativamente el tiempo.
Un aspecto crucial de las máquinas cuánticas es que la lectura del resultado modifica el estado de los qubits, porque la mecánica cuántica establece que cualquier interacción con el mundo subatómico produce en él un cambio. La idea general para resolver este problema consiste en garantizar que el algoritmo cuántico realice los cómputos necesarios antes de destruir lo que se conoce como la *coherencia entre estados superpuestos*, además de contar con un mecanismo adecuado de corrección de errores.

La clase de problemas que pueden ser resueltos por algoritmos cuánticos en tiempo polinomial, con una probabilidad de error menor que 1/4 (ó 1/3, o cualquier valor real ε que cumpla $0 < \varepsilon < 1/2$), se denomina BQP (por *Bounded error, Quantum, Polynomial time*). Esta clase incluye problemas de interés de los que no se conocen soluciones polinomiales utilizando máquinas convencionales.
BQP incluye a las clases P y BPP, y está incluida en las clases PP y PSPACE. Por lo tanto:
$P \subseteq BPP \subseteq BQP \subseteq PP \subseteq PSPACE$.
No se sabe si las inclusiones son estrictas. Tampoco se conoce la relación entre BQP y NP, y se conjetura que BQP no incluye a los problemas NP-completos.

Dos áreas en las que se ha incursionado con los algoritmos cuánticos son la criptografía y el análisis de grandes volúmenes de información.
Por ejemplo, el algoritmo cuántico de Shor para factorizar números trabaja en tiempo polinomial. Lo importante de este resultado es que rompe con la eficiencia de los mejores métodos criptográficos de la actualidad, que pasarían a ser obsoletos si el algoritmo de Shor pudiera ser ejecutado en una computadora cuántica real (si un número grande es el producto de dos números primos de aproximadamente el mismo tamaño, no existe algoritmo convencional conocido capaz de factorizarlo en tiempo polinomial; los mecanismos actuales de la criptografía consideran este hecho).
Otro problema difícil relacionado con la criptografía es el problema del logaritmo discreto. Tanto este problema como el anterior, en sus formas de problemas de decisión, pertenecen a la clase BQP, y no parecieran estar en P.

También se destaca el algoritmo cuántico de Grover. Resuelve velozmente las búsquedas en bases de datos desordenadas. Mientras que por medio de los algoritmos convencionales, la localización de un dato en una base de datos desordenada de tamaño N requiere en promedio N/2 búsquedas, el algoritmo de Grover trabaja en tiempo $O(\sqrt{N})$.

Referencias bibliográficas de la Parte II

El estudio de la complejidad computacional tuvo fundamentalmente sus inicios en los trabajos de Hartmanis y Stearns. Los teoremas de jerarquía fueron probados en (Hartmanis y Stearns, 1965). Los teoremas de gaps se desarrollan en (Borodin, 1972). La jerarquía polinomial aparece en (Stockmeyer, 1977), y tiene una formulación equivalente mediante máquinas de Turing alternantes (Chandra, Kozen y Stockmeyer, 1981). Meyer y Stockmeyer presentaron el primer problema que requería tiempo (y espacio) exponencial (Meyer y Stockmeyer, 1973). Blum definió la teoría de la complejidad en forma axiomática (Blum, 1967). El artículo (Cook, 1982) describe la evolución de la complejidad computacional en sus primeros 20 años, en oportunidad de haber recibido el autor el premio Turing de ese año.

La tesis de Cobham-Edmonds fortalece la tesis de Church-Turing, estableciendo que todo par de modelos de computación razonables y generales se encuentran polinomialmente relacionados. En la primera parte de este libro desarrollamos la simulación de una máquina de Turing con K cintas por medio de una máquina con una cinta, y dicha simulación elevaba de manera cuadrática el tiempo de ejecución. Por ejemplo, en (Kushilevitz y Nisan, 1996) se demuestra que pasar de K cintas a una cinta incrementa el tiempo al cuadrado. En cambio, pasar de K cintas a dos cintas incrementa el tiempo sólo en un factor logarítmico (Hennie y Stearns, 1966).

Los trabajos de Cook (Cook, 1971), Karp (Karp, 1972) y Levin (Levin, 1973) constituyeron el comienzo de la teoría de NP-completitud. El libro (Garey y Johnson, 1979) es un excelente compendio de problemas NP-completos. Si bien es posible definir problemas completos de una manera natural para muchas clases de complejidad (Cai, 2003), hemos optado por introducir problemas en la clase NPC a partir del lenguaje SAT, que representa el problema de satisfactibilidad de las fórmulas booleanas. El Teorema de Cook, que prueba que SAT es NP-completo, apareció por primera vez en (Cook, 1971). La reducción polinomial del lenguaje SAT al lenguaje 3-SAT puede encontrarse en (Cai, 2003). El algoritmo polinomial para resolver el problema 2-SAT aparece en (Gács y Lovász, 1999). Con respecto a la completitud en general, recomendamos la lectura de (Schaefer y Umans, 2002), en que en el mismo estilo de Garey y Johnson se compendian problemas completos del segundo y tercer nivel de la jerarquía polinomial.

La conjetura de Hartmanis-Berman establece que todos los lenguajes NP-completos son p-isomorfos. Por otro lado, la conjetura de Joseph-

Young establece la existencia de un lenguaje NP-completo tal que no existe una reducción poly-time invertible desde SAT a dicho lenguaje (Cai, 2003). En (Ladner, 1975-a), además de probarse la existencia de problemas en la clase NPI (suponiendo que $P \neq NP$), se muestra que si $P \neq NP$, entonces NPI debe contener pares de lenguajes C y D tales que ni $C \alpha_P D$ ni $D \alpha_P C$. Esto quiere decir que de acuerdo al orden de dificultad (en el sentido de la complejidad computacional) impuesto en NP por la relación α_P, deberán existir lenguajes incomparables si se cumple la conjetura $P \neq NP$. Adleman y Manders muestran la existencia de problemas con la propiedad de estar en P sii NP = CO-NP (Adleman y Manders, 1977). Berman demostró que si un lenguaje tally está en la clase NPC, entonces P = NP (Berman, 1978). Posteriormente, Mahaney demostró lo mismo pero con respecto a los lenguajes esparsos (Mahaney, 1982). Otra demostración de este último teorema también puede encontrarse en (Cai, 2003). Entre los numerosos trabajos sobre la relación entre P y NP, recomendamos en especial el breve artículo (Gasarch, 2002), en que los más importantes representantes del área de la complejidad computacional responden a una encuesta referida a P vs NP: cuándo se resolverá, cómo se resolverá, y qué ténicas se utilizarán.

La prueba de que el pasaje del no determinismo al determinismo impacta en un orden cuadrático con respecto al espacio, aparece en (Savitch, 1970). Que el espacio no determinístico es cerrado con respecto al complemento fue demostrado en forma independiente por Szelepcsényi (Szelepcsényi, 1987) y por Immerman (Immerman, 1988). Es inmediato que se cumple $DTIME(f(n)) \subseteq DSPACE(f(n))$, y en (Hopcroft, Paul y Valiant, 1975) se muestra que si la función f es tiempo-construible, vale la inclusión $DTIME(f(n)) \subseteq DSPACE(f(n) / \log f(n))$.

Los trabajos (Ausiello, D'Atri y Protasi, 1980) y (Paz y Moran, 1981) son fundacionales en el marco de los algoritmos de aproximación y la optimización de problemas que se mencionaron en el Apéndice 2. En (Bruschi, Joseph y Young, 1991) se muestran diversos resultados relacionados con el tema.

Baker, Gill y Solovay introdujeron la relativización de P y NP mediante el uso de oráculos (Baker, Gill y Solovay, 1975). Otro tratamiento del tema fue realizado por Pratt y Stockmeyer mediante máquinas vectoriales (Pratt y Stockmeyer, 1976). En (Sipser, 1992) se describen varios de los trabajos realizados con el objetivo de llegar a una respuesta sobre la relación entre P y NP.

En (de Leeuw, Moore, Shannon y Shapiro, 1956) y (Von Neumann, 1961) se describen las máquinas de Turing probabilísticas. Las clases probabilísticas mencionadas en el Apéndice 2, como BPP y ZPP, aparecen en (Gill, 1977). Se recomienda la lectura de (Motwani y

Raghavan, 1995) y (Mitzenmacher y Upfal, 2005) para el estudio de algoritmos probabilísticos.

Las maquinas de Turing pueden recibir cualquier input y de cualquier tamaño. Existen otros modelos computacionales donde una máquina sólo es apta para recibir inputs de una determinada longitud. Tal es el caso de los circuitos booleanos. Dichos modelos resultan útiles para determinar cotas inferiores de complejidad computacional. Shannon introdujo los circuitos booleanos en (Shannon, 1938). La clase NC y los algoritmos paralelos se abordan por ejemplo en (Leighton, 1992). Un compendio de problemas P-completos puede encontrarse en (Greenlaw, Hoover y Ruzzo, 1992).

Los sistemas de pruebas interactivas fueron definidos en forma independiente en (Babai y Moran, 1988) y (Goldwasser, Micali y Rackoff, 1989). La prueba de IP = PSPACE se desarrolla en (Shamir, 1992).

Feynman percibió la posibilidad de utilizar la mecánica cuántica para darle más potencia a las máquinas de Turing clásicas (Feynman, 1982). En 1985, Deutsch definió una primera versión de las máquinas de Turing cuánticas (Deutsch, 1985). Posteriormente se precisaron las definiciones (Deutsch y Jozsa, 1992), (Yao, 1993) y (Bernstein y Vazirani, 1997). El lector interesado puede además consultar los libros (Nielsen y Chuang, 2000) y (Griffiths, 2003), y el artículo (Ambainis, 2004) en que se pasa revista a los algorítmicos cuánticos de búsqueda más importantes.

Los algoritmos con los que hemos trabajado debían tener en cuenta los casos donde el input podía no estar en una forma válida para el problema a resolver. Even, Selman y Yacobi definieron la noción de *promise problems* (Even, Selman y Yacobi, 1984), según la cual un algoritmo para resolver un promise problem no necesita contemplar los casos en que el input no es válido. Goldreich analiza y muestra algunas aplicaciones de dicha noción (Goldreich, 2005). Otra noción consiste en suponer que contamos con el valor de determinada función aplicada a una propiedad del input, por ejemplo la función «es primo» aplicada a la longitud del input. Usualmente esto se modeliza dotando a las máquinas de Turing con un input de sólo lectura, denominado *advice*, que se basa en una función aplicada a la longitud del input original. En este contexto, las máquinas de Turing reciben el nombre de máquinas de Turing con advice. Fueron introducidas en (Karp y Lipton, 1980). Un resultado interesante sobre dichas máquinas se logra analizando las máquinas con advice que trabajan en tiempo determinístico polinomial, y restringiendo las funciones de advice a las que son computables en tiempo determinístico polinomial. Esta clase de complejidad usualmente se denomina P/Poly; en (Ladner, 1975-b) se la relaciona con la jerarquía espacial y los circuitos booleanos.

Un área específica de la teoría de la complejidad se basa en medir la longitud de la máquina de Turing (u otro modelo computacional equivalente) más pequeña para generar un string. De esta manera es posible analizar la cantidad de información codificada por un string. A esta área se la conoce como teoría de la complejidad de Kolmogorov (Kolmogorov, 1965), aunque también fue definida en forma independiente en (Solomonoff, 1964) y (Chaitín, 1969). En (Li y Vitányi, 1990) se encuentran aplicaciones de dicha teoría. En (Kolmogorov, 1992) se describe la relación entre la teoría de la complejidad de Kolmogorov y las clases de complejidad P y NP.

Artículos y libros recientes que recomendamos leer por los temas que abordan son (Immerman, 2005), en que se repasa el status de la complejidad descriptiva, caracterizada por relacionar la complejidad computacional con la descripción de los problemas utilizando la lógica; (Guo y Niedermeier, 2007), en que se profundiza la técnica de kernelización, mediante la cual el primer paso para resolver un problema difícil es simplificar en tiempo polinomial las instancias del problema, de modo tal de trabajar luego sólo con un kernel representativo de los datos; y (Dietzfelbinger, 2004), libro dedicado completamente a la demostración de que el problema de primalidad está en P. Dicho teorema, de agosto de 2002, fue resuelto por M. Agrawal, N. Kayal y N. Saxena. El trabajo recibió atención mundial inmediata, no sólo porque resolvía una pregunta abierta por espacio de 30 años, sino además porque las técnicas utilizadas fueron elementales.

A continuación se lista la que consideramos bibliografía principal en relación a la segunda parte del libro. En este sentido, (Bovet y Crescenzi, 1994) y (Papadimitriou, 1995) tratan en conjunto varios de los temas que hemos desarrollado:

- Adleman L. y Manders K. [1977]. Reducibility, randomness and intractability. Proc. Ninth Annual ACM Symposium on the Theory of Computing, pp. 151-163.
- Aho A. V., Hopcroft J. E. y Ullman J. D. [1974]. The design and analysis of computer algorithms. Addison-Wesley.
- Ambainis A. [2004]. Quantum Search Algorithms. SIGACT News, 33, 2, 34-48.
- Arora S. y Barak B. [2007]. Computational Complexity: A Modern Approach. Princeton University.
- Ausiello G., D'Atri A. y Protasi M. [1980]. Structure preserving reduction among convex optimization problems. Journal of Computer and System Sciences, 21, 136-153.
- Babai L. y Moran S. [1988]. Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes. Journal of Computer and System Sciences, 36(2) : 254–276.

- Baker T., Gill J. y Solovay R. [1975]. Relativizations of the P =? NP question. SIAM Journal on Computing, 4(4):431–442.
- Balcázar J. L., Díaz J. y Gabarró J. [1990]. Structural Complexity I. Monographs in Theoretical Computer Science. An EATCS Series. An EATCS Series. Springer-Verlag.
- Balcázar J. L., Díaz J. y Gabarró J. [1995]. Structural Complexity I, 2nd, rev. ed. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag.
- Baum G. [1987]. Complejidad. Kapelusz, EBAI.
- Berman P. [1978]. Relationship between density and deterministic complexity of NP-complete languages. In Proceedings of the 5th Conference on Automata, Languages and Programming, pp.63–71. Springer-Verlag. Lecture Notes in Computer Science 62.
- Bernstein E. y Vazirani U. [1997]. Quantum complexity theory. SIAM Journal on Computing, 26(5):1411–1473.
- Blum M. [1967]. A Machine-Independent Theory of the Complexity of Recursive Functions. Journal of the ACM. Vol. 14 (2), pages 290-305.
- Borodin A. [1972]. Computational Complexity and the Existence of Complexity Gaps. Journal of the ACM. Vol. 19 (1), pages 158-174.
- Bovet D. P. y Crescenzi P. [1994]. Introduction to the Theory of Complexity. Prentice- Hall.
- Bruschi D., Joseph D. y Young P. [1991]. A structural overview of NP optimization problems. Algorithms Reviews, 2, 1-26.
- Cai Jin-Yi [2003]. Lectures in Computational Complexity. University of Winconsin.
- Chaitín G. J. [1969]. On the length of programs for computing finite binary sequences: statistical considerations. J. Assoc. Comp. Mach. 16:145-159.
- Chandra A. K., Kozen D. C. y Stockmeyer L. J. [1981]. Alternation. Journal of the ACM, 28(1):114-133.
- Cobham A. [1964]. The intrinsic computational difficulty of functions. Proc. Congress for Logic, mathematics and Philosophy of Science, 24-30.
- Cook S. A. [1971]. The Complexity of Theorem Proving Procedures. In 3rd ACM Symposium on the Theory of Computing, pages 151-158.
- Cook S. A. [1982]. An Overview of Computational Complexity. ACM Turing Award Lectures. The first twenty years 1966-1985. ACM Press.
- de Leeuw K., Moore E. F., Shannon C. E. y Shapiro N. [1956]. Computability by probabilistic machines. In C. E. Shannon and J. McCarthy, editors, Automata Studies, pages 183–212.
- Deutsch D. [1985]. Quantum theory, the Church-Turing principle and the universal quantum computer. Proceedings of the Royal Society of London Ser. A, A400:97–117.
- Deutsch D. y Jozsa R. [1992]. Rapid solution of problems by quantum computation. Proc Roy Soc Lond A, 439 : 553–558.
- Dietzfelbinger, M. [2004]. Primality Testing in Polynomial Time. Springer-Verlag.
- Dijkstra E. W. [1959]. A note on two problems in connection with graphs. Numerische Mathematik, 1, 269-271.

- Edmonds J. R. [1965]. Paths, trees and flowers. Canadian Journal of Mathematics, 17, 449-467.
- Even S., Selman A. L. y Yacobi Y. [1984]. The Complexity of Promise Problems with Applications to Public-Key Cryptography. Information and Control, Vol. 61, pages 159-173.
- Feynman R. [1982]. Simulating physics with computers. International Journal of Theoretical Physics, 21(6&7):467–488.
- Gács P. y Lovász L. [1999]. Complexity of Algorithms, Lecture Notes.
- Garey M. R. y Johnson D. S. [1979]. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, New York.
- Gasarch W. [2002]. The P =? NP Poll. SIGACT News, 33, 2, 34-48.
- Gill J. [1977]. Computational complexity of probabilistic Turing machines. SIAM Journal on Computing, 6(4):675–695.
- Goldreich O. [2005]. On Promise Problems (a survey in memory of Shimon Even). Weizmann Institute of Science.
- Goldreich O. [2008]. Computational Complexity: A Conceptual Perspective. Cambridge University Press.
- Goldwasser S., Micali S. y Rackoff C. [1989]. The knowledge complexity of interactive proof systems. SIAM Journal on Computing, 18(1):186–208.
- Greenlaw R., Hoover H.J. y Ruzzo W. L. [1992]. A compendium of problems complete for P. University of Alberta, Computer Science Department, Technical Report 91-11.
- Griffiths R. B. [2003]. Consistent Quantum Theory. Cambridge University Press.
- Guo J. y Niedermeier R. [2007]. Invitation to data reduction and problem kernelization. SIGACT News, 38, 1, March 2007.
- Hartmanis J. y Stearns R.E. [1965]. On the Computational Complexity of Algorithms. Transactions of the AMS. Vol. 117 pages 285-306.
- Håstad J. [1999]. Complexity Theory. Royal Institute of Thechnology.
- Hennie F.C. y Stearns R.E. [1966]. Two-tape simulation of multitape Turing machines. Journal of the ACM, 13(4):533-546.
- Hopcroft J. E., Paul W. J. y Valiant L. G. [1975]. On time vs space and related problems. Proc. 16th Annual IEEE Symp. on the Foundations of Computer Science, pp. 57-64.
- Hopcroft J. E. y Ullman J. D. [1979]. Introduction to automata theory, languages and computation. Addison-Wesley.
- Immerman N. [1988]. Nondeterministic Space is Closed Under Complementation. SIAM Journal on Computing, Vol. 17, pages 760-778.
- Immerman N. [2005]. Progress in Descriptive Complexity. SIGACT News, 36, 4, December 2005.
- Karp R. M. [1972]. Reducibility among Combinatorial Problems. In Complexity of Computer Computations, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, pp 85-103.
- Karp R. M. y Lipton R. J. [1980]. Some connections between nonuniform and uniform complexity classes. In 12th ACM Symposium on the Theory of Computing. pp. 302-309.

- Knuth D. E. [1968]. The art of computer programming, vol. 1: fundamental algorithms. Addison-Wesley.
- Kolmogorov A. N. [1965]. Three approaches to the quantitative definition of information. Prob. of Information Transmission 1, pp. 1-7.
- Kolmogorov A. N. [1992]. Kolmogorov Complexity and Computational Complexity. EATCS Monographs on Theoretical Computer Science. Springer-Verlag Berlin Heidelberg.
- Kozen D. C. [1992]. The design and analysis of algorithms. Springer-Verlag.
- Kushilevitz E. y Nisan N. [1996]. Communication Complexity. Cambridge University Press.
- Ladner R. E. [1975-a]. On the structure of polynomial time reducibility. Journal of the ACM, 22(1):155-171.
- Ladner R. E. [1975-b]. The circuit value problem is log space complete for P. ACM SIGACT News 7:1, pp 18-20.
- Ladner R. E. y Goldberg J. [2000]. A pedagogical Proof of the Immerman-Szelepscényi Theorem. University of Washington.
- Leighton F. T. [1992]. Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann.
- Levin L. A. [1973]. Universal Search Problems. Problemy Peredachi Informatsii 9, pp. 115-116.
- Li M., Vitányi P. M. B. [1990]. Kolmogorov complexity and its applications. The Handbook of Theoretical Computer Science, vol. I: Algorithms and Complexity, MIT Press, pp. 187-254.
- Mahaney S. R. [1982]. Sparse complete sets for NP: Solution of a conjecture by Berman and Hartmanis. Journal of Computer and System Sciences 25:130-143.
- Meyer A. R. y Stockmeyer L. J. [1973]. The equivalence problem for regular expressions with squaring requires exponential space. Proc. Thirteenth Annual IEEE Symposium on Switching and Automata Theory, pp. 125-129.
- Mitzenmacher M. y Upfal E. [2005]. Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press.
- Motwani R. y Raghavan P. [1995]. Randomized algorithms. Cambridge University Press.
- Nielsen M. y Chuang I. [2000]. Quantum Computation and Quantum Information. Cambridge University Press.
- Papadimitriou C. M. [1995]. Computational complexity. Addison-Wesley.
- Paz A. y Moran S. [1981]. Non deterministic polynomial optimization problems and their approximation. Theoretical Computer Science, 15, 251-277.
- Pippenger N. [1979]. On simultaneous resource bounds. Proc. IEEE Symposium on Foundations of Computer Science, 307-311.
- Pratt V. R. y Stockmeyer L. J. [1976]. A characterization of the power of vector machines. J. Computer and System Sciences 12:2, 198-221.
- Rabin M. O. [1960]. Degree of difficulty of computing a function and a partial ordering of recursive sets. Technical Report 2, Hebrew University, Jerusalem.
- Ruzzo W. L. [1981]. On uniform circuit complexity. Journal of Computer and System Sciences, 22, 365-383.

- Savage J. E. [1972]. Computational work and time on finite machines. Journal of the ACM, 19(4):660–674.
- Savitch W. J. [1970]. Relationships between nondeterministic and deterministic tape complexities. J. Computer and System Sciences 4:2, 177-192.
- Schaefer M. y Umans C. [2002]. Completeness in the polynomial-time hierarchy. SIGACT News, 33, 3 y 4, September - December 2002.
- Seiferas J. J. [1990]. Machine-independent complexity theory. Van Leeuwen J. ed., handbook of theoretical computer science, vol. A, Elsevier, 163-186.
- Shamir A. [1992]. IP = PSPACE. Journal of the ACM, 39(4):869–877.
- Shannon C. E. [1938]. A Symbolic Analysis of Relay and Switching Circuits. Trans. American Institute of Electrical Engineers, Vol. 57, pages 713-723.
- Sipser M. [1992]. The history and status of the P versus NP question. In ACM, editor, Proceedings of the twenty-fourth annual ACM Symposium on Theory of Computing, Victoria, British Columbia.
- Solomonoff R. [1964]. A formal theory of inductive inference, Part. I. Information and Control 7:1-22.
- Solovay R. y Strassen V. [1977]. A fase Monte-Carlo test for primality. SIAM Journal on Computing, 6, 84-85.
- Stockmeyer L. J. [1977]. The Polynomial-Time Hierarchy. Theoretical Computer Science. Vol. 3, pages 1-22.
- Szelepcsényi R. [1987]. The method of forcing for nondeterministic automata. Bull. of the EATCS, 33, pp. 96-100.
- Trevisan L. [2004]. Lecture Notes on Computational Complexity, Computer Science Division, U.C. Berkeley.
- Valiant L. G. [1979]. The Complexity of Computing the Permanent. Theoretical Computer Science, Vol. 8, pp. 189-291.
- Vishkin U. [1984]. On the choice of a model of parallel computation. Manuscript.
- Von Neumann J. [1961]. Probabilistic logics and synthesis of reliable organisms from unreliable components, volume 5. Pergamon Press.
- Yao A. [1993]. Quantum circuit complexity. In 34th Annual Symposium on Foundations of Computer Science IEEE, pp. 352–361, Palo Alto, California.

Ejercicios de la Parte II

E.II.1 Sea la siguiente definición alternativa a la utilizada en el libro:

$[f(n) = O(g(n))] \leftrightarrow [\exists c > 0$ y $\exists n_0 \in N$ tales que $\forall n \in N: n \geq n_0 \rightarrow f(n) \leq c.g(n)]$.

Probar que las dos definiciones son equivalentes.

E.II.2 Dadas dos clases de lenguajes recursivos C_1 y C_2, probar que $C_1 \subseteq C_2$ sii CO-$C_1 \subseteq$ CO-C_2, siendo CO-$C_i = \{L \varepsilon R \mid L^C \varepsilon C_i\}$, i = 1, 2.

E.II.3 Sea f una función perteneciente a P, y sea L un lenguaje perteneciente a P. Se define:

$f^{-1}(L) = \{x \mid f(x) \varepsilon L\}$.

Probar que $f^{-1}(L)$ también pertenece a P.

E.II.4 Proponer una manera de enumerar todas las máquinas de Turing (MT) que trabajan en tiempo exactamente T(n), siendo T una función tiempo-construible.

E.II.5 Demostrar que las siguientes funciones son tiempo-construibles:

a) n^2 b) 2^n c) n!

E.II.6 Determinar cuáles de los siguientes lenguajes pertenecen a P:

a) SMALL-SAT = $\{\vartheta \mid \vartheta$ es una fórmula booleana en la forma normal conjuntiva, y existe una asignación de valores de verdad que la satisface en la que hay a lo sumo 10 variables con valor de verdad verdadero$\}$.

b) 2-COL= {G | G es un grafo 2-coloreable}. Un grafo G es 2-coloreable si existe una función f de los vértices de G al conjunto {0,1}, tal que $f(v) \neq f(v')$ si (v, v') es un arco de G.

c) SPATH = {(G, a, b, K) | el grafo G tiene un camino de longitud a lo sumo K entre los vértices a y b}.

d) LPATH = {(G, a, b, K) | el grafo G tiene un camino de longitud al menos K entre los vértices a y b}.

E.II.7 Probar que toda fórmula booleana f puede transformarse en tiempo determinístico polinomial en una fórmula booleana g en la forma normal conjuntiva, tal que f es satisfactible sii g es satisfactible.

E.II.8 Una fórmula booleana es una tautología si es satisfecha por cualquier asignación. Dar un algoritmo determinístico de tiempo polinomial para decidir si una fórmula booleana es una tautología.

E.II.9 Una fórmula booleana de Horn es una fórmula booleana en la forma normal conjuntiva, tal que cada cláusula tiene a lo sumo una variable no negada. Dar un algoritmo determinístico de tiempo polinomial para decidir si una fórmula booleana de Horn es satisfactible.

E.II.10 Probar que el problema CSAT restringido a que cada variable pueda aparecer sólo dos veces, se puede resolver en tiempo determinístico polinomial.

E.II.11 Probar que el problema 3-SAT restringido a que cada variable pueda aparecer a lo sumo tres veces, se puede resolver en tiempo determinístico polinomial.

E.II.12 Construir una reducción poly-time del lenguaje CSAT al lenguaje 3-SAT.

E.II.13 Probar que el lenguaje {0,1} es P-completo con respecto a las reducciones poly-time.

E.II.14 Sea M_1 una MT no determinística tal que $L(M_1) = L$, y M_1 trabaja en tiempo polinomial. Sea la siguiente MTN M_2: trabaja como M_1, pero cuando M_1 acepta entonces M_2 rechaza, y cuando M_1 rechaza entonces M_2 acepta. ¿Por qué esto no prueba que NP = CO-NP?

E.II.15 Sea M_1 una MTN, tal que para todo string w de $L(M_1)$, alguna computación de M_1 que acepta w tarda $O(n^k)$ pasos, con $n = |w|$ y k constante. Construir una MTN M_2 equivalente a M_1 que trabaje en tiempo polinomial.

E.II.16 Probar que la clase NP es cerrada con respecto a la unión y la intersección.

E.II.17 Probar que los siguientes lenguajes pertenecen a la clase NP:

a) DOM-SET = {(G, K) | el grafo G tiene un conjunto dominante de K nodos}. Un subconjunto de vértices de un grafo G es un conjunto dominante de G, si todo otro vértice de G es adyacente a algún vértice de dicho subconjunto.

b) ISO = {(G_1, G_2) | G_1 y G_2 son grafos isomorfos}. Dos grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ son isomorfos si $V_1 = V_2$ y existe

una permutación Π de 1, 2, …, m, siendo m la cantidad de vértices de V_1, tal que para todo par (i, k), $(v_i, v_k) \in E_1 \leftrightarrow (v_{\Pi(i)}, v_{\Pi(k)}) \in E_2$. En palabras, G_1 y G_2 son isomorfos si son iguales a no ser por la identificación de sus arcos.

E.II.18 Sea el problema que plantea, dados dos grafos G_1 y G_2, si acaso G_1 tiene un circuito de Hamilton y G_2 no. Definir el lenguaje asociado al problema, y determinar si pertenece a NP o a CO-NP.

E.II.19 Una MT no determinística fuerte es una MT no determinística tal que cada una de sus computaciones puede terminar en el estado de aceptación («sí»), de rechazo («no») o de duda («puede ser»). Una MT M de este tipo decide un lenguaje L si se cumple lo siguiente: si w ε L, entonces todas las computaciones de M terminan en el estado «sí» o el estado «puede ser», y al menos una en el estado «sí». Si en cambio w $\notin$ L, entonces todas las computaciones de M terminan en el estado «no» o el estado «puede ser», y al menos una en el estado «no». Probar que un lenguaje L es decidido por una MT no determinística fuerte en tiempo polinomial sii L ε NP $\cap$ CO-NP.

E.II.20 Probar que el lenguaje HP pertenece al conjunto NPH.

E.II.21 Sea el lenguaje $L = \{0^k 1 0^h \mid k \text{ y } h \text{ son números naturales mayores que cero}\}$. Probar que se cumple L ε NPC $\leftrightarrow$ P = NP.

E.II.22 Probar que si se cumple $L_1 \alpha_P L_2$ y $L_2 \alpha_P L_1$, y además $L_1 \varepsilon$ NPC, entonces $L_2 \varepsilon$ NPC.

E.II.23 Se define, dada una fórmula booleana ϑ, la función $C(\vartheta)$, que calcula el número de asignaciones que satisfacen ϑ. A su vez, una aproximación $\mathbb{A}$ de una función f con respecto a una cota D ≥ 1, es una función que cumple que $\forall x$: $f(x) / D \leq \mathbb{A}(x) \leq f(x).D$. Probar que si existe una aproximación $\mathbb{A} \varepsilon$ P de C con respecto a una cota D ≥ 1, entonces P = NP.

E.II.24 En cada uno de los casos siguientes determinar si la propiedad se cumple, siendo A y B lenguajes no triviales, es decir, distintos de los conjuntos $\varnothing$ y Σ^*:

a) Si B $\in$ P, entonces A $\cap$ B es reducible en tiempo polinomial a A.

b) Si B $\in$ P, entonces A $\cup$ B es reducible en tiempo polinomial a A.

c) Si A es NP-completo, $A^C \in$ NP y B $\in$ NP, entonces $B^C \in$ NP.

d) Si A es NP-completo, $A^C \in$ NP y B $\in$ NP, entonces $B^C \in$ P.

e) Si A $\cap$ B es NP-completo, A $\in$ NP y B $\in$ P, entonces A es NP-completo.

f) Si A $\cup$ B es NP-completo, A $\in$ NP y B $\in$ P, entonces A es NP-completo.

E.II.25 Sea el lenguaje L = {(<M>, w, t) | M acepta w en a lo sumo t pasos}. Probar que L es NP-completo.

E.II.26 Probar que es NP-completo el lenguaje asociado al problema de determinar si, dado un grafo G con un circuito de Hamilton, G tiene otro circuito de Hamilton.

E.II.27 Demostrar que cada uno de los siguientes lenguajes es NP-completo. En cada caso identificar la naturaleza del problema que dificulta encontrar un algoritmo determinístico polinomial:

a) Dado un grafo G = (V,E), un conjunto V' $\subseteq$ V se dice independiente sii para todo par de vértices s, t de V', se cumple que (s, t) $\notin$ E. El lenguaje IS se define de la siguiente manera: IS = {(G, K) | K es un número natural y G es un grafo que tiene un conjunto independiente de K vértices}.

b) Dado un grafo G = (V,E), se dice que G tiene un camino de Hamilton si existen s, t $\in$ V tales que G tiene un camino entre s y t que recorre todos los vértices restantes una sola vez. El lenguaje HPA se define de la siguiente manera: HPA = {G | G es un grafo que tiene un camino de Hamilton}.

E.II.28 Se pide lo mismo que en el ejercicio anterior, pero antes se debe definir el lenguaje asociado al problema que se formula:

a) Dado un conjunto de n conjuntos, ¿acaso existen k conjuntos, con k < n, cuya unión es la misma que la unión de los n conjuntos?

b) Dado un grafo G = (V, E) y un número entero positivo k, determinar la existencia de un subconjunto V' de V, con |V'| < k, tal que todo circuito en G incluya al menos un vértice de V'.

c) Dado un grafo G = (V, E), determinar si G tiene un subconjunto V' de V tal que: (i) V' es un conjunto independiente, y (ii) para todo vértice v $\in$ V – V', existe al menos un arco entre v y algún vértice de V'.

E.II.29 Probar que si $S_1(n) = O(S_2(n))$, entonces NSPACE($S_1(n)$) $\subseteq$ NSPACE($S_2(n)$).

E.II.30 Probar que si $L \in NTIME(T(n))$, entonces existe una MTN M que trabaja en espacio T(n) y L(M) = L. ¿Vale la recíproca?

E.II.31 Sea FPSPACE el conjunto de las funciones totales $f: \Sigma^* \rightarrow N$ computables determinísticamente en espacio polinomial. Por otro lado, sea #FP el conjunto de las funciones totales $f: \Sigma^* \rightarrow N$ tales que $f \in$ #FP sii existe una MTN M que trabaja en tiempo polinomial y a todo input w lo acepta en exactamente f(w) computaciones. Probar que se cumple que $\#FP \subseteq FPSPACE$.

E.II.32 Sean $A \in NLOGSPACE$ y $B \in NP$. Demostrar que $B - A \in NP$.

E.II.33 ¿Se cumple que si NP = PSPACE, entonces NP = CO-NP?

E.II.34 ¿Se cumple que $PSPACE \neq DTIME(n)$?

E.II.35 Probar que si S(n) > log(n) es una función espacio-construible, entonces existe un lenguaje recursivo L que no pertenece a DSPACE(S(n)).

E.II.36 Determinar la relación entre cada uno de los siguientes pares de clases de complejidad:

a) $DSPACE(n^2)$ y DSPACE(f(n)), con f(n) = n para n impar, n^3 para n par.

b) $NSPACE(2^n)$ y $DSPACE(5^n)$.

E.II.37 ¿Qué relación existe entre la clase SPACE(O(1)) y la clase de los lenguajes regulares?

E.II.38 Probar que si todo lenguaje de NP es log-space reducible a un lenguaje de P, entonces se cumple P = NP.

E.II.39 ¿Se cumple que si todo lenguaje de NP es poly-time reducible a un lenguaje de DLOGSPACE, entonces DLOGSPACE = NP?

E.II.40 Sea el problema de determinar si un grafo orientado G es conexo, es decir, si entre cualquier par de vértices del grafo orientado G existe un camino. Probar que el lenguaje asociado a dicho problema es NLOGSPACE-completo con respecto a las reducciones log-space.

E.II.41 Probar que los enunciados siguientes son equivalentes:

a) A es Turing-reducible a B.

b) A^C es Turing-reducible a B.

c) A es Turing-reducible a B^C.

d) A^C es Turing-reducible a B^C.

E.II.42 Se definió una MT probabilística de tipo PP como una MTN que trabaja en tiempo polinomial y que acepta w sii más de la mitad de sus computaciones termina en el estado q_A. El problema 1/2 SAT se formula de la siguiente manera: dada una fórmula booleana ϑ, ¿acaso ϑ es satisfecha por más de la mitad de las posibles asignaciones de valores de verdad? Se pide construir una MT de tipo PP que decida el problema 1/2 SAT.

E.II.43 Siendo PP la clase de lenguajes aceptados por las MT probabilísticas de tipo PP, probar que NP $\subseteq$ PP.

PARTE III
VERIFICACIÓN DE PROGRAMAS

Introducción de la Parte III

La última escala de nuestro viaje imaginario se sitúa aún más cerca de los problemas decidibles. En este último tramo no seremos meros espectadores, sino que tendremos un comportamiento más activo: trataremos de asegurar que los algoritmos que se escriban para resolver los problemas sean correctos…

En esta última parte del libro, nuestra mirada a los problemas decidibles se basa en un punto de vista totalmente distinto, pero tan fundamental como el de la complejidad computacional. Se trata de la correctitud de los algoritmos, término que de ahora en más vamos a reemplazar por *programas*, escritos en lenguajes de programación específicos, y con un nivel de abstracción mucho mayor que el de las instrucciones de una máquina de Turing.
En efecto, la tercera parte del libro trata sobre la *verificación de programas*. La idea es plantear una metodología para probar la correctitud de un programa con respecto a un conjunto de propiedades que debe satisfacer, denominado especificación. Se presentan de una manera sistemática, métodos de verificación para programas secuenciales (determinísticos y no determinísticos) y concurrentes. La familia de programas con la que se trabaja es muy representativa: los programas imperativos. Estos programas se caracterizan por transformar estados a través de las instrucciones que los componen.
En un programa secuencial, el control reside en todo momento en un solo lugar. Las propiedades a demostrar serán: la *correctitud parcial*, es decir que si el programa termina, la relación entre el input y el output debe ser correcta según lo establecido por la especificación; y la *termi-*

nación. La terminación no sólo significa finitud, sino también que el programa *no falle*: por ejemplo, que no tenga una división por cero, un índice fuera de rango, una selección condicional sin condiciones verdaderas, etc.

Un programa concurrente se compone de varios procesos secuenciales, que se ejecutan concurrentemente y se comunican y sincronizan entre sí. El control reside al mismo tiempo en varios lugares, cada uno correspondiente a un proceso secuencial distinto. Además de la correctitud parcial y la terminación, se considerará la prueba de *ausencia de deadlock*, es decir que no exista ningún proceso que quede bloqueado indefinidamente a la espera de un evento que nunca se va a producir. Otras dos propiedades a las que se hará referencia son: la *ausencia de inanición* (o *ausencia de starvation*), es decir que todo proceso que compita por un recurso (acceso a memoria, ejecución de una instrucción, etc.) alguna vez lo obtenga; y la *exclusión mutua*, es decir que ningún proceso manipule inadecuadamente variables que comparte con otro proceso.

Una aproximación natural para la verificación de programas es la *operacional*, cuyo análisis se realiza en términos de las computaciones asociadas a los programas. Obviamente, esta aproximación resulta prohibitiva cuando se aplica a programas concurrentes muy complejos, con numerosas computaciones y propiedades a probar.

Otra aproximación, muy difundida en la literatura y que es la que seguiremos en este libro, es la *axiomática*. Se utiliza un lenguaje de especificación para expresar formalmente las propiedades que se deben satisfacer. Se describe formalmente la semántica del lenguaje de programación utilizado. Y como en la lógica, se define un método de prueba, con axiomas y reglas asociadas en este caso a las instrucciones del lenguaje de programación, que permite demostrar la correctitud de un programa con respecto a su especificación, de la misma forma en que se prueba un teorema. Por lo tanto, a diferencia de la aproximación operacional, las pruebas de la aproximación axiomática son sintácticas, y se basan en la técnica de inducción sobre la estructura del programa.

El tipo de semántica que se utiliza para describir los lenguajes de programación es la *semántica operacional*: las descripciones se hacen en términos de las operaciones de una máquina abstracta. Otra opción podría ser la *semántica denotacional*, que tiene un enfoque mucho más matemático, y cuyo uso se torna muy complejo para el caso de los programas concurrentes. Hay un capítulo que trata resumidamente la semántica denotacional en el Apéndice 3. Como lenguaje de especificación se utiliza lá *lógica de primer orden con igualdad*, que se interpreta en todos los casos sobre el dominio de los números enteros.

Además de analizar métodos para probar propiedades de programas, también se analizan propiedades de los propios métodos. Las dos propiedades que se van a considerar son: la *sensatez*, que se cumple si toda fórmula probada dentro del método es verdadera; y la *completitud*, que es la inversa de la sensatez, es decir que se cumple si toda fórmula verdadera puede ser probada dentro del método.

Los pioneros de la aproximación axiomática fueron R. Floyd en 1967, considerando los diagramas de flujo, y C. Hoare en 1969, ya sobre programas secuenciales determinísticos del tipo Pascal, con repeticiones while. Posteriormente, E. Dijkstra en 1975 extendió la metodología de Hoare a los programas secuenciales no determinísticos, mientras que S. Owicki y D. Gries hicieron lo propio en 1976 y 1977 con respecto a los programas concurrentes con variables compartidas, y K. Apt, N. Francez y W. de Roever en 1980 con respecto a los programas concurrentes con pasajes de mensajes (o programas distribuidos).

Si bien los ejemplos que se presentan consisten en pruebas a posteriori, es decir que se exponen pruebas de programas ya construidos con respecto a especificaciones existentes, la idea subyacente es mostrar principios generales de construcción de programas correctos. En otras palabras, los axiomas y reglas de los métodos de verificación pueden verse como guías para el desarrollo sistemático de programas.

Se tratan solamente programas de *entrada/salida*. Queda fuera del alcance de este libro la verificación de los programas *reactivos*, en los que no existe la noción de terminación sino de interacción permanente con el entorno de ejecución (es el caso, por ejemplo, de un sistema operativo). De todos modos, hay un capítulo en el Apéndice 3 que se refiere a este tema de una manera resumida, y empleando lógica temporal.

La metodología de verificación de programas se introduce incrementalmente, empezando con los programas secuenciales y terminando con los programas concurrentes. El objetivo no es meramente enumerar métodos y desarrollar ejemplos de aplicación, sino discutir conceptos que se repiten y amplían a lo largo de los capítulos, de acuerdo a las características de los programas estudiados.

En el Capítulo 3.1 se introducen definiciones fundamentales para el tratamiento de la verificación de programas, como especificación, programa, semántica operacional, correctitud parcial y total, etc. Para facilitar la formulación de estos conceptos, se consideran inicialmente sólo los programas secuenciales determinísticos.

En el Capítulo 3.2 se trata la verificación de los programas secuenciales determinísticos. En estos programas, la siguiente instrucción se determina unívocamente. Primero se presenta el lenguaje de

programación (sintaxis y semántica operacional). Luego se describen los métodos de verificación para la correctitud parcial y la correctitud total (correctitud parcial más terminación). La separación en dos métodos se debe a que la correctitud parcial y la terminación se prueban mediante técnicas distintas. Los conceptos más importantes relacionados con estas pruebas son las aserciones invariantes, los variantes decrecientes (o funciones cota), y la composicionalidad. Posteriormente se trata la sensatez y la completitud de los métodos.
Para facilitar la presentación, se trabaja solamente con variables simples, si bien se analiza el impacto en la sensatez de los métodos cuando se introducen los arreglos. Finalmente se presentan las *proof outlines* (esquemas de prueba), que son una forma alternativa de presentar las pruebas, imprescindibles en la verificación de los programas concurrentes. También se hace mención al desarrollo sistemático de programas sobre la base de los métodos presentados.
En el Capítulo 3.3 se extiende la metodología a los programas secuenciales no determinísticos. Como existe no determinismo en la concurrencia, este capítulo también sirve como introducción para el próximo. Ahora puede haber más de una instrucción siguiente posible, y por lo tanto varias computaciones y estados finales. La estructura de la presentación de los temas es la misma que la del capítulo anterior. Se mantiene la composicionalidad de los métodos de verificación.
La novedad en este caso es el concepto de *fairness* (justicia o equidad). Con fairness, una instrucción habilitada frecuentemente para ser ejecutada a lo largo de una computación infinita, no puede postergarse indefinidamente. El fairness impacta sobre la terminación de los programas, y por lo tanto el método de verificación de correctitud total debe considerarlo.
En el Capítulo 3.4 se extiende la metodología a los programas concurrentes. Se considera tanto el modelo con variables compartidas como el modelo con pasajes de mensajes, en términos de distintos lenguajes de programación. En cada caso, se mantiene la presentación de los temas de los capítulos anteriores. Otra vez, dado un programa y un estado inicial, puede haber varias computaciones y varios estados finales, producto de la semántica de intercalación (*interleaving*) de las instrucciones atómicas de los procesos. Como la generación de las posibles secuencias de instrucciones atómicas es no determinística, se debe seguir considerando la hipótesis de fairness.
Una de las novedades en este marco, es que se amplía el conjunto de propiedades a probar, básicamente con la ausencia de deadlock. Otra novedad es que los métodos dejan de ser composicionales, lo que determina que las pruebas deban efectuarse en dos etapas. En la

primera etapa se prueban los procesos secuenciales utilizando proof outlines, y en la segunda etapa se consisten las proof outlines según un criterio de consistencia que depende del modelo de concurrencia y el lenguaje de programación. Una tercera característica distintiva de la verificación de programas concurrentes es la necesidad de introducir variables auxiliares e invariantes globales, para que los métodos de verificación sean completos.

Finalmente, el Apéndice 3 trata resumidamente la verificación de programas secuenciales determinísticos con procedimientos, la verificación de programas reactivos utilizando lógica temporal (incluyendo una mención a la verificación automática de programas de estados finitos utilizando lógica temporal proposicional), y la semántica denotacional.

CAPÍTULO 3.1. DEFINICIONES INICIALES

En este capítulo se definen los conceptos fundamentales necesarios para encarar la verificación de distintos tipos de programas. Para simplificar esta introducción, se consideran primero solamente los programas determinísticos secuenciales. Por lo tanto, las definiciones se irán enriqueciendo a lo largo de los capítulos.

En la Sección 3.1.1 se introducen nociones tales como especificación, lenguaje de especificación, programa y semántica de un lenguaje de programación. Como la familia de programas con que se va a trabajar es la de los programas imperativos, se formaliza además el concepto de estado.

El comportamiento de los programas se explica en términos de estados, configuraciones y computaciones. En este sentido, volvemos a recurrir a las elementales máquinas de Turing, ahora para definir operacionalmente la semántica de los lenguajes de programación.

No se profundiza en la problemática de la especificación de programas, esto queda fuera del alcance del libro. El foco está puesto en la prueba de correctitud de un programa con respecto a una especificación ya existente. Por tal motivo, se utiliza un lenguaje de especificación muy sencillo, la lógica de primer orden, que facilita la presentación de los temas. Las propiedades que debe satisfacer un programa se especifican como pares de aserciones lógicas, una para representar el input y la otra para representar el output. Es decir que sólo se va a considerar el comportamiento de entrada/salida de los programas (por ejemplo, se ignoran propiedades relacionadas con el tiempo y el espacio consumidos). No se tratan los programas reactivos, en los que no hay noción de terminación, porque estos programas interactúan permanentemente con el entorno de ejecución. En este caso, la lógica temporal es adecuada para especificar las propiedades que deben satisfacer. En el Capítulo A3.2 del Apéndice 3 se hace referencia a este tema.

En la Sección 3.1.2 se describe cómo se va a tratar la prueba de correctitud de un programa con respecto a una especificación. Se introducen las primeras propiedades que se van a considerar en las pruebas: la correctitud parcial y la terminación. La correctitud parcial no asegura la terminación, sólo establece que si el programa termina, produce un output adecuado con respecto al input y a la especificación del programa. La correctitud parcial y la terminación se tratan por separado porque se prueban con técnicas distintas. Un programa será totalmente correcto con respecto a una especificación si es parcialmente correcto y termina con respecto a la misma.

Consecuentemente, se definen dos métodos de verificación, uno para la prueba de correctitud parcial y el otro para la prueba de correctitud total, que es una extensión del anterior considerando la prueba de terminación. En la misma sección se prueba el Lema de Separación, que fundamenta la separación de la prueba de correctitud total en dos pruebas.
Finalmente se describe la forma de los métodos de verificación, la cual se basa en los sistemas deductivos de la lógica. Las pruebas son sintácticas, consideran axiomas y reglas relacionadas con las instrucciones del lenguaje de programación. Se definen dos propiedades que los métodos debieran cumplir: la sensatez, es decir que lo que prueben se cumpla semánticamente, y la propiedad inversa, la completitud, es decir que lo que se cumpla semánticamente pueda probarse dentro de los métodos.

3.1.1. Especificaciones y programas

Verificar un programa consiste en probar que el programa, escrito en un determinado lenguaje de programación, es correcto con respecto a una especificación. La especificación establece, mediante un determinado lenguaje de especificación, propiedades que el programa debe cumplir. Para verificar el programa con respecto a la especificación, es necesario describir la semántica del lenguaje de programación. Por lo tanto, un método de verificación de programas se define en términos de la sintaxis y semántica de un lenguaje de especificación y de un lenguaje de programación. En lo que sigue, se desarrollan estos conceptos.
Primeramente, es necesario definir qué es un *estado*. Un estado es una asignación de valores a variables, una «instantánea» de los contenidos de las variables de un programa. Informalmente, dadas las variables x, y, z, un ejemplo de estado podría ser:

x = 5, y = verdadero, z = «a».

Formalmente, y considerando solamente variables de tipo entero:

Definición. Un estado es una función

$$\sigma: Ivar \rightarrow V,$$

tal que Ivar es un conjunto de variables enteras, y V es el conjunto de los números enteros.

□

La expresión $\sigma(x)$ denota el contenido de la variable x. El conjunto de todos los estados σ de Ivar a V se denota con:

$$\Sigma = (\text{Ivar} \rightarrow V).$$

Utilizando el concepto de estado, la siguiente es una primera definición de *programa*, que se refinará enseguida: un programa puede ser visto como una función f: $\Sigma \rightarrow \Sigma$, o en otras palabras, como un transformador de estados, desde un estado inicial $\sigma \varepsilon \Sigma$ hasta un estado final $\sigma' \varepsilon \Sigma$. Un programa se escribe en un determinado lenguaje de programación, y su comportamiento se describe mediante la semántica de dicho lenguaje. El tipo de semántica que se considerará en este libro es la *semántica operacional*. Formalmente:

Definición. La semántica operacional de un lenguaje de programación Stat, describe mediante una máquina abstracta, cómo un programa $S \varepsilon$ Stat transforma estados a partir de un estado inicial $\sigma \varepsilon \Sigma$.

□

Más precisamente, dados el programa S y el estado inicial σ, la idea es asociar a la configuración inicial $C_0 = <S, \sigma>$ una computación π , que es una secuencia de transformaciones de configuraciones $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots$ Una configuración $C_i = <S_i , \sigma_i>$ está compuesta por una continuación sintáctica S_i , que informalmente se define como lo que falta por consumirse del programa S para alcanzar la continuación sintáctica vacía E, y por un estado corriente σ_i . Las computaciones pueden ser finitas o infinitas (también se dice, respectivamente, que las computaciones pueden converger o diverger). La computación finita $C_0 \rightarrow \ldots \rightarrow C_k$ se puede denotar con $C_0 \rightarrow^* C_k$. La configuración final $C_k = <E, \sigma_k>$ de una computación finita se denomina configuración terminal. Una computación π a partir de $C_0 = <S, \sigma>$ se denota con $\pi(S, \sigma)$, o bien $\pi(C_0)$, y se describe formalmente mediante una relación binaria «$\rightarrow$» que se denomina relación de transición. El estado final de la computación $\pi(S, \sigma)$ se denota con val($\pi(S, \sigma)$). Si $\pi(S, \sigma)$ termina en el estado σ_k, se escribe val($\pi(S, \sigma)$) = σ_k (se dice que σ_k es un estado propio). Si, en cambio, $\pi(S, \sigma)$ no termina, se escribe val($\pi(P, \sigma)$) = $\perp$. El símbolo $\perp$ denota una computación infinita, y se lo conoce como estado indefinido.

Una computación siempre es maximal, en el sentido de que o bien termina en una configuración terminal o bien es infinita.
En este libro no se profundizará sobre la semántica de los lenguajes de programación, sólo se utilizará lo indispensable para estudiar distintos métodos de verificación de programas. Para cada uno de los lenguajes de programación que se presenten en los siguientes capítulos, se definirá su semántica operacional. En el Apéndice 3, como complemento, se hace una suscinta referencia a otro tipo de semántica, la semántica *denotacional*, que tiene un enfoque mucho más matemático que la semántica operacional. Podría encararse el estudio de la verificación de programas también mediante esta semántica alternativa.
Con las consideraciones anteriores, ahora se puede refinar la definición anterior de programa, introduciendo la definición de *función semántica*:

Definición. Dado un lenguaje de programación Stat,

$$\mathcal{M} : \text{Stat} \rightarrow (\Sigma \rightarrow \Sigma), \text{ tal que } \mathcal{M}(S)(\sigma) = \text{val}(\pi(S, \sigma)),$$

es la función semántica asociada al lenguaje Stat.

□

Es decir, un programa es una función $\mathcal{M}(S)$: $\Sigma \rightarrow \Sigma$, que transforma estados iniciales en estados finales. Dicho de otra manera, la función semántica $\mathcal{M}$ asigna a un par formado por un programa S, que es un objeto sintáctico, y un estado inicial $\sigma \in \Sigma$, que es un objeto semántico, el valor $\text{val}(\pi(S, \sigma))$, otro objeto semántico, el cual puede ser un estado propio $\sigma' \in \Sigma$, o bien el estado indefinido $\perp$. Por lo tanto, $\mathcal{M}$ es una función total. En lo que sigue, se usarán en forma indistinta las expresiones $\mathcal{M}(S)(\sigma)$ y $\text{val}(\pi(S, \sigma))$.
La relación entre los estados iniciales y finales de un programa se establece por medio de una *especificación* del programa. El lenguaje de especificación que se utilizará en este libro es la lógica de primer orden con igualdad (interpretada sobre el dominio de los números enteros), y se lo denominará Assn o lenguaje de aserciones. Formalmente:

Definición. $\Phi = \langle p, q \rangle$ es una especificación de un programa S, tal que:

- p es una aserción de la lógica de primer orden, que denota el conjunto de estados iniciales de S. Se denomina precondición.

- q es una aserción de la lógica de primer orden, que denota el conjunto de estados finales de S. Se denomina postcondición.
- El par <p, q> establece la relación que debe existir entre los estados iniciales (o inputs) y los estados finales (u outputs) de S.

□

Las aserciones p y q se expresan en términos de las variables de programa, que aparecen como variables libres. Como se verá más adelante en algunos ejemplos, también se pueden utilizar otras variables. De esta manera, además de la visión semántica de los programas como transformadores de estados, también existe la visión sintáctica de los programas como transformadores de aserciones.
La expresión:

$$\sigma \models p,$$

denota que el estado σ satisface la aserción p (p es verdadero cuando se evalúa con los contenidos de σ). O en otras palabras, el estado σ pertenece al conjunto de estados denotado por la aserción p.
Se define:

- $\sigma \models$ true, para todo estado $\sigma \in \Sigma$. Es decir, true denota el conjunto de todos los estados.
- $\sigma \not\models$ false, para todo estado $\sigma \in \Sigma$. Es decir, false denota el conjunto vacío de estados.
- Además, por convención, $\perp \not\models p$, para toda aserción $p \in$ Assn.

La especificación de programas es una tarea fundamental en el proceso de desarrollo de software, y su tratamiento está fuera del alcance de este libro. El foco estará puesto en la prueba de correctitud de un programa con respecto a una especificación existente. En lo que resta de esta sección, se presentan algunos ejemplos relacionados con la problemática de la especificación de programas. La sección siguiente, que complementa este capítulo introductorio, menciona las características generales de los métodos de verificación programas.
El siguiente es un primer ejemplo de especificación de un programa:

Ejemplo. Sea Φ = <x = X, x = 2X>.

Se cumple que un programa S satisface la especificación Φ si duplica su input x.

En el ejemplo, el conjunto de estados iniciales, denotado por la aserción p = (x = X), es todo Σ. Si por ejemplo *y* es una variable del programa S, el hecho de no mencionarla en la precondición significa que inicialmente puede tener cualquier valor. Por su parte, el conjunto de estados finales, denotado por la postcondición q = (x = 2X), es tal que la variable x tiene un número entero par.
Notar que se indica el valor inicial de x en términos de X, para poder establecer su valor final después de la ejecución de S, dado que x puede cambiar. La variable x es una variable de programa. La variable X se denomina variable de especificación, no forma parte del conjunto de variables del programa S, e implícitamente está cuantificada universalmente. A diferencia de x, el valor de X no cambia.
Las variables de especificación se utilizan para «congelar» valores (en este caso, el valor inicial de x). En el capítulo del Apéndice 3 dedicado a la verificación de programas con procedimientos, se verá más acerca de la utilidad de estas variables.
El siguiente ejemplo muestra el uso de la aserción true:

Ejemplo. Se pretende especificar un programa denominado SORT, tal que al final se cumpla la condición x > y.

Una posible especificación del programa SORT es Φ = <true, x > y>.

Ejercicio. *Comparar la especificación anterior Φ = <x = X, x = 2X> con la especificación Φ' = <true, ∃Y: x = 2Y>, siendo Y una variable de especificación. ¿Acaso las dos especificaciones son equivalentes?*

□

Finalmente, el ejemplo que se presenta a continuación muestra cómo aún en casos sencillos, se pueden plantear erróneamente las especificaciones:

Ejemplo. Se pretende especificar un programa S ε Stat, tal que al final se cumpla la condición y = 1 ó y = 0, según al comienzo valga o no, respectivamente, la propiedad p(x), dada una variable de programa x. Se asume la existencia en el lenguaje de programación Stat, de la instrucción de asignación x := e (asignación de una expresión entera e,

a la variable entera x), y de la instrucción de secuencia, denotada con el operador «**;**».

Una primera versión de la especificación, errónea, sería:

Φ_1 = <true, (y = 1 $\rightarrow$ p(x)) $\wedge$ (y = 0 $\rightarrow \neg$ p(x))>.

Notar que el programa S :: y := 2, satisface Φ_1 pero no es el programa que se pretende especificar.

Una segunda versión de la especificación, también errónea, sería:

Φ_2 = <true, (0 $\leq$ y $\leq$ 1) $\wedge$ (y =1 $\rightarrow$ p(x)) $\wedge$ (y = 0 $\rightarrow \neg$ p(x))>.

Sea el programa S :: x := 5 ; y := 1. Notar que si al comienzo, el valor de x es 7, no se cumple p(7), y se cumple p(5), entonces el programa S satisface Φ_2, y así otra vez, S no es el programa que se pretende especificar.

Finalmente, el siguiente intento resulta exitoso:

Φ_3 = <x = X, (0 $\leq$ y $\leq$ 1) $\wedge$ (y = 1 $\rightarrow$ p(X)) $\wedge$ (y = 0 $\rightarrow \neg$ p(X))>.

Ejercicio. *Modificar la especificación del ejemplo, para que además establezca que el programa S preserve el valor inicial de la variable x.*

□

3.1.2. Métodos de verificación de programas

Los métodos de verificación de programas se van a presentar, en los próximos capítulos, asumiendo que las pruebas de correctitud parten de la existencia a priori de una especificación y de un programa (es decir, son pruebas a posteriori). De todos modos, la idea subyacente es describir principios fundamentales para la construcción de programas correctos. Dicho de otro modo, el foco está puesto en la construcción de un programa en simultáneo con su prueba de correctitud con respecto a una especificación (ver la figura 3.1.1).

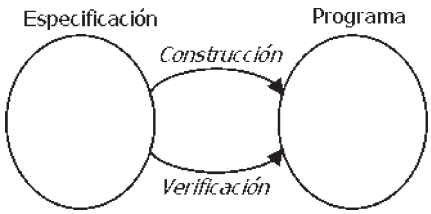


Figura 3.1.1

Se presentan por separado métodos de verificación para dos tipos de propiedades: la *correctitud parcial* de programas y la *correctitud total* de programas, que se definen a continuación:

Definición. Un programa S es *parcialmente correcto* con respecto a la especificación $\Phi = <p, q>$, sii para todo estado $\sigma \varepsilon \Sigma$:

$$(\sigma \models p \wedge val(\pi(S, \sigma)) \neq \perp) \rightarrow val(\pi(S, \sigma)) \models q.$$

□

En palabras, el programa S es parcialmente correcto con respecto a la especificación $\Phi = <p, q>$, o directamente <p, q>, sii 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 propiedad de 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. Tampoco se viola la propiedad cuando se consideran estados σ que no satisfacen p, independientemente de lo que suceda al final de las computaciones correspondientes.
La expresión $\models \{p\}\ S\ \{q\}$ denota que S es parcialmente correcto con respecto a <p, q>. La fórmula de correctitud {p} S {q} se conoce como terna de Hoare de correctitud parcial.

Ejemplo. Todo programa S cumple $\models \{true\}\ S\ \{true\}$. En palabras, a partir de cualquier estado, todo programa S, si termina, lo hace en algún estado.

La prueba es la siguiente. Sean $\sigma \varepsilon \Sigma$ y S ε Stat. Debe cumplirse:

$$(\sigma \models \text{true} \wedge \text{val}(\pi(S, \sigma)) \neq \perp) \rightarrow \text{val}(\pi(S, \sigma)) \models \text{true}.$$

Se cumple $\sigma \models$ true. Si $\text{val}(\pi(S, \sigma)) = \perp$, entonces se cumple la implicación trivialmente. Y si $\text{val}(\pi(S, \sigma)) \neq \perp$, entonces $\text{val}(\pi(S, \sigma)) = \sigma' \models$ true, por lo que también en este caso se cumple la implicación.

□

Por otro lado, si se cumple $\models$ {true} S {false}, entonces significa que S no termina a partir de ningún estado. La prueba es la siguiente. Sean $\sigma \in \Sigma$ y S $\in$ Stat. Debe cumplirse:

$$(\sigma \models \text{true} \wedge \text{val}(\pi(S, \sigma)) \neq \perp) \rightarrow \text{val}(\pi(S, \sigma)) \models \text{false}.$$

Se cumple $\sigma \models$ true. Si $\text{val}(\pi(S, \sigma)) \neq \perp$, entonces $\text{val}(\pi(S, \sigma)) = \sigma' \models$ false (absurdo). Por lo tanto $\text{val}(\pi(S, \sigma)) = \perp$, es decir, el programa S no termina a partir de σ.

□

Por su parte, la correctitud total se define de la siguiente manera:

Definición. Un programa S es *totalmente correcto* con respecto a <p, q>, sii para todo estado $\sigma \in \Sigma$:

$$\sigma \models p \rightarrow (\text{val}(\pi(S, \sigma)) \neq \perp \wedge \text{val}(\pi(S, \sigma)) \models q).$$

□

Es decir, el programa S es totalmente correcto con respecto a la especificación <p, q>, sii a partir de cualquier estado σ que satisface la precondición p, S termina en un estado σ' que satisface la postcondición q. La correctitud total de S con respecto a <p, q> no se viola cuando se consideran estados σ que no satisfacen p, independientemente de lo que suceda al final de las computaciones asociadas.
La expresión $\models$ <p> S <q> denota que S es totalmente correcto con respecto a <p, q>. La fórmula de correctitud <p> S <q> se conoce como terna de Hoare de correctitud total.
Por lo tanto, la correctitud total reúne dos requisitos básicos que debe cumplir un programa S con respecto a una especificación <p, q>:

a) S debe transformar los estados iniciales en los estados finales, según lo establecido por la especificación <p, q>.
b) S debe terminar a partir de todos los estados iniciales que satisfacen la precondición p.

Otra forma de expresar lo mismo es que la correctitud total es la correctitud parcial más la terminación (en realidad, más adelante se introducen otras propiedades, por lo que la correctitud parcial más la terminación serán solamente parte de la correctitud total). La figura 3.1.2 ilustra esta idea. Σ_1 es el conjunto de estados iniciales, denotado por una determinada precondición p, y Σ_2 es el conjunto de estados finales, denotado por una determinada postcondición q. A partir de cualquier estado $\sigma_1 \varepsilon \Sigma_1$, S debe terminar en un estado $\sigma_2 \varepsilon \Sigma_2$. Por otro lado, no se establece ninguna condición para S a partir de un estado $\sigma_3 \notin \Sigma_1$.

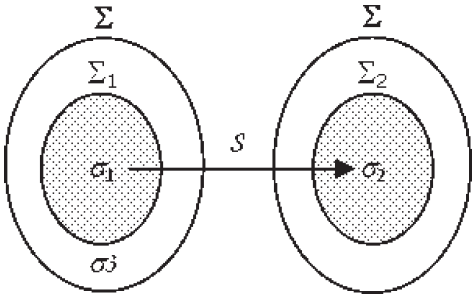


Figura 3.1.2

El siguiente lema fundamenta el modo en que se van a presentar, en los ejemplos, las pruebas de correctitud total de los programas:

Lema de Separación. Se cumple que $\models \langle p \rangle S \langle q \rangle \leftrightarrow (\models \{p\} S \{q\} \wedge \models \langle p \rangle S \langle true \rangle)$.

Prueba. Primero se probará $\models \langle p \rangle S \langle q \rangle \rightarrow (\models \{p\} S \{q\} \wedge \models \langle p \rangle S \langle true \rangle)$:

Sea $\models \langle p \rangle S \langle q \rangle$. Dado σ, vale $\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \wedge val(\pi(S, \sigma)) \models q)$. Por lo tanto:
a) $(\sigma \models p \wedge val(\pi(S, \sigma)) \neq \perp) \rightarrow val(\pi(S, \sigma)) \models q$, es decir $\models \{p\} S \{q\}$.
b) $\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \wedge val(\pi(S, \sigma)) \models true)$, es decir $\models \langle p \rangle S \langle true \rangle$.
Entonces, por (a) y (b): $\models \{p\} S \{q\} \wedge \models \langle p \rangle S \langle true \rangle$.

Ahora se probará $(\models \{p\} S \{q\} \wedge \models \langle p \rangle S \langle true \rangle) \rightarrow \models \langle p \rangle S \langle q \rangle$:

Sea $\models \{p\}\ S\ \{q\} \wedge \models \langle p\rangle\ S\ \langle true\rangle$. Dado σ, vale:
a) $(\sigma \models p \wedge val(\pi(S, \sigma)) \neq \perp) \rightarrow val(\pi(S, \sigma)) \models q$.
b) $\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \wedge val(\pi(S, \sigma)) \models true)$.
Por (b): $\sigma \models p \rightarrow val(\pi(S, \sigma)) \neq \perp$.
Por (a) y (b): $\sigma \models p \rightarrow val(\pi(S, \sigma)) \models q$.
Entonces: $\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \wedge val(\pi(S, \sigma)) \models q)$, es decir $\models \langle p\rangle\ S\ \langle q\rangle$.
□

La separación entre métodos de verificación de correctitud parcial y métodos de verificación de correctitud total no es caprichosa, se debe a que las pruebas se basan en distintas técnicas en uno y en otro caso. La correctitud parcial se prueba por inducción, mientras que la terminación se prueba utilizando algún conjunto parcialmente ordenado bien fundado (es decir, algún conjunto parcialmente ordenado estricto sin cadenas descendentes infinitas), como por ejemplo el conjunto de los números naturales con la relación «<».
En los capítulos siguientes se describe cómo probar éstas y otras propiedades. Se va a considerar una clasificación en dos grandes familias de propiedades, las de tipo *safety* (seguridad), como la correctitud parcial, que se prueban inductivamente en base a aserciones invariantes, y las de tipo *liveness* (vivacidad), como la terminación, que se prueban en base a variantes decrecientes de conjuntos parcialmente ordenados bien fundados.
Los componentes de un método de verificación de programas son los propios de un sistema deductivo. Para probar una fórmula de correctitud $\{p\}\ S\ \{q\}$ (lo mismo vale para una fórmula de correctitud $\langle p\rangle\ S\ \langle q\rangle$), se utiliza un conjunto finito de axiomas y un conjunto finito de reglas, ámbos relacionados con las instrucciones del lenguaje de programación asociado. El propósito es construir una prueba de la fórmula $\{p\}\ S\ \{q\}$.
La expresión:

$$\vdash_D \{p\}\ S\ \{q\},$$

denota que $\{p\}\ S\ \{q\}$ se prueba en el método de verificación D. La prueba tiene la forma:

$$(1) \vdash_D f_1,\ (2) \vdash_D f_2,\ (3) \vdash_D f_3,\ \ldots,\ (k) \vdash_D f_k,$$

siendo $f_k = \{p\}\ S\ \{q\}$, y cualquier f_i un axioma o una fórmula que se obtiene de $f_1, f_2, f_3, \ldots, f_{i-1}$ por empleo de una o más reglas. En términos lógicos, $\{p\}\ S\ \{q\}$ es un teorema del sistema deductivo D.

Es decir, la prueba es sintáctica, se basa en axiomas y reglas que establecen cómo las instrucciones del lenguaje de programación transforman aserciones. Esto se conoce como *aproximación axiomática*. Notar que la *aproximación operacional*, basada en pruebas semánticas que manipulan estados, cuando los dominios son infinitos (por ejemplo para el caso del conjunto de los números enteros), sólo pueden asegurar la presencia de errores, nunca su ausencia.
Así planteadas las pruebas, debe asegurarse que por medio de un método de verificación no se puedan demostrar ternas de Hoare que no valgan semánticamente, es decir que no sean verdaderas (por ejemplo {x = 0} x := 1 {x = 2}, asumiendo la semántica habitual de la instrucción de asignación). Formalmente, se debe probar la *sensatez* (*soundness*) del método de verificación:

Definición. Un método D para la verificación de la correctitud parcial de programas S es *sensato* (*sound*) sii, para todo programa S y para toda especificación <p, q>, se cumple:

$$\vdash_D \{p\}\ S\ \{q\} \rightarrow \models \{p\}\ S\ \{q\}.$$

□

La sensatez de un método para la verificación de la correctitud total de programas se define de una manera similar.

Ejercicio. *Si D es un método sensato, y se cumple tanto* $\vdash_D \{p\}\ S\ \{q\}$ *como* $\vdash_D \{p\}\ S\ \{\neg q\}$*, ¿qué se puede decir del programa S con respecto a la precondición p?*

□

También se va a requerir la propiedad recíproca, es decir, que por medio de un método de verificación pueda probarse cualquier terna de Hoare verdadera (como {x = 0} x := 1 {x = 1}). Formalmente, se debe probar la *completitud* del método de verificación:

Definición. Un método D para la verificación de la correctitud parcial de programas S es *completo* sii, para todo programa S y para toda especificación <p, q>, se cumple:

$$\models \{p\}\ S\ \{q\} \rightarrow \vdash_D \{p\}\ S\ \{q\}.$$

□

La completitud de un método para la verificación de la correctitud total de programas se define de una manera similar.

CAPÍTULO 3.2. VERIFICACIÓN DE PROGRAMAS SECUENCIALES DETERMINÍSTICOS

En este capítulo se introduce detalladamente la metodología de verificación de programas. La forma de la presentación de los temas se va a repetir en los siguientes capítulos.
Se consideran los programas secuenciales determinísticos de entrada/salida. El control del programa está en todo momento en un solo lugar, y la instrucción siguiente se define unívocamente. A partir de un estado inicial, la ejecución del programa produce una sola computación, que puede terminar en un estado final o no terminar.
En la Sección 3.2.1 se introduce el lenguaje de programación con el que se va a trabajar, el lenguaje PLW de los while, similar a cualquier lenguaje tipo Pascal. Se define su sintaxis y su semántica operacional. A partir de la definición inductiva de la semántica, luego se explicitan las distintas formas que pueden tener las computaciones de los programas de PLW. En el Capítulo A3.3 del Apéndice 3 se brinda más detalle de la sintaxis y semántica de PLW, llegando al nivel de las expresiones enteras y booleanas, pero utilizando un tipo de semántica alternativo, la semántica denotacional.
En las siguientes dos secciones se describen los métodos de verificación de correctitud parcial y correctitud total de los programas de PLW. Una característica esencial de los mismos es la *composicionalidad*: si el programa S está compuesto por los subprogramas S_1 y S_2, entonces la correctitud de S depende sólo de la correctitud de S_1 y S_2, sin necesidad de referirse a la estructura interna de los subprogramas.
El uso de *aserciones invariantes*, para la prueba de correctitud parcial, y de *variantes decrecientes* de conjuntos parcialmente ordenados bien fundados, para la prueba de terminación, caracterizan las dos técnicas utilizadas para las pruebas. La prueba de correctitud parcial es inductiva (la inducción se conoce como computacional, porque se aplica sobre las computaciones de los programas). En cambio, en la prueba de terminación no hay noción de propiedad invariante; por el contrario, se debe definir un variante decreciente (o *función cota*) que alcance alguna vez en el transcurso del while asociado, el valor mínimo de una secuencia decreciente de números naturales.
En la Sección 3.2.2 se describe el método H para la verificación de la correctitud parcial de los programas de PLW. Se enumeran sus axiomas y reglas, se presentan ejemplos de aplicación, y se prueba la sensatez y la completitud del método.

Para simplificar la presentación no se consideran estructuras de datos complejas. Tampoco se extiende el lenguaje PLW con procedimientos. De todos modos se plantea la problemática de la sensatez del método H cuando se introducen arreglos. Y en el Capítulo A3.1 del Apéndice 3 se tratan resumidamente extensiones al método cuando PLW incluye procedimientos con recursión y parámetros.

La prueba detallada de la completitud del método H queda fuera del alcance de este libro. Se remarca, no obstante, la problemática del poder expresivo del lenguaje de especificación, necesario para denotar mediante aserciones de la lógica de primer orden todos los conjuntos de estados intermedios de una computación. En particular, se hace mención a la expresividad requerida para denotar los invariantes de los while. Se introduce el concepto de completitud *relativa*, producto del teorema de incompletitud de Gödel y, justamente, de la dependencia del poder expresivo del lenguaje de especificación utilizado, en el que impacta entre otras cosas la variedad de operadores del sublenguaje de las expresiones enteras. El tratamiento de la completitud de los métodos de verificación será menor en los siguientes capítulos.

En la Sección 3.2.3 se describe el método H* para la verificación de la correctitud total de los programas de PLW. Los axiomas y reglas son obviamente los mismos de H, salvo la regla asociada a la instrucción de repetición while, la única que puede producir no terminación. Los ejemplos de aplicación de H* se basan en el Lema de Separación, es decir: para probar la correctitud total de un programa con respecto a la especificación <p, q>, se prueba mediante el método H la correctitud parcial del programa con respecto a <p, q>, y mediante H* la correctitud total con respecto a <p, true>. También se hace referencia a la sensatez y completitud relativa de H*.

Finalmente, en la Sección 3.2.4 se agregan temas misceláneos. Se describen reglas redundantes para el método H que acortan las pruebas. Se presenta una forma alternativa de exponer las pruebas, denominadas proof outlines (esquemas de prueba). La idea es intercalar las fórmulas de correctitud de la prueba entre las instrucciones del programa, obteniendo una forma más estructurada, y produciendo además el efecto colateral de documentar el programa adecuadamente. Para la verificación de los programas concurrentes, las proof outlines se tornan componentes fundamentales de la metodología de prueba. También se hace mención al desarrollo sistemático de programas.

3.2.1. Lenguaje de programación PLW

Para desarrollar los temas de este capítulo, el lenguaje de programación genérico Stat referido anteriormente se va a instanciar en el lenguaje PLW (lenguaje de programación con instrucciones while). PLW es un subconjunto muy simple de cualquier lenguaje tipo Pascal.

La sintaxis de un programa S ε PLW es la siguiente:

S :: skip | x := e | S_1 ; S_2 | if B then S_1 else S_2 fi | while B do S_1 od

La semántica de las instrucciones de PLW es la habitual. Informalmente:

La instrucción skip es atómica y «no hace nada». La instrucción de asignación x := e también es atómica, y consiste en la asignación del valor entero asociado a la expresión entera e, a la variable entera x. Ejemplos de expresiones enteras son: 5, y, z + 8, etc.
El resto de las instrucciones se componen de instrucciones más elementales (se van a definir enseguida), y son: la secuencia o composición secuencial (denotada con el operador «**;**»), la selección condicional, y la repetición. B es una expresión booleana, denominada guardia (o guarda) booleana, que en la instrucción while se evalúa antes de la ejecución del cuerpo S_1 de la repetición. Ejemplos de expresiones booleanas son: true, $x < y$, $x = 0 \vee z = 0$, etc.

Ejemplo. El siguiente programa de PLW implementa la especificación <x > 0, y = x!>, es decir, calcula en y el factorial de x, con x > 0.

S_{fac} :: a := 1 ; y := 1 ;
while a < x do a := a + 1 ; y := y **.** a od

La semántica (operacional) formal de PLW se define inductivamente por medio de la relación de transición «$\rightarrow$» introducida en el capítulo anterior.
En este caso, «$\rightarrow$» describe una máquina abstracta que paso a paso transforma configuraciones de la forma C = <S_i, σ_i >, siendo S_i una continuación sintáctica de un programa S ε PLW, y σ_i el estado corriente.
La semántica formal del lenguaje de programación PLW es la siguiente:

(1) $\langle skip, \sigma \rangle \rightarrow \langle E, \sigma \rangle$.

La instrucción skip «no hace nada», es decir, no modifica el estado corriente.
La expresión E denota la continuación sintáctica vacía, y no pertenece al lenguaje PLW. Para todo S ε PLW se cumple que: S ; E = E ; S = S.

(2) $\langle x := e, \sigma \rangle \rightarrow \langle E, \sigma[x \mid e] \rangle$.

La instrucción de asignación modifica eventualmente el estado corriente.
La notación $\sigma[x \mid e]$ abrevia $\sigma[x \mid \sigma(e)]$. El estado $\sigma[x \mid e]$ es idéntico al estado σ, salvo que el contenido de la variable entera x es el valor entero que resulta de evaluar la expresión entera e en el estado σ. Formalmente:

Definición. La *variante* del estado σ con respecto a la variable entera x, y al número entero a, es decir el estado $\sigma[x \mid a]$, se define de la siguiente manera:

a) Si x = y, entonces $\sigma[x \mid a](y) = a$.
b) Si $x \neq y$, entonces $\sigma[x \mid a](y) = \sigma(y)$.

□

Notar que la expresión previa $\sigma(e)$ es un abuso de notación, dado que σ es una función de Ivar a V y e es una expresión entera. Para simplificar la presentación, no se va a definir la semántica operacional más allá del nivel de las instrucciones del lenguaje PLW. En el capítulo del Apéndice 3 relacionado con la semántica denotacional, se consideran las definiciones semánticas de las expresiones.

(3) Si $\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle$, entonces para todo T ε PLW: $\langle S ; T, \sigma \rangle \rightarrow \langle S' ; T, \sigma' \rangle$.

Notar que como S ; E = E ; S = S, entonces si $val(\pi(S, \sigma)) = \sigma'' \neq \perp$, vale: $\langle S ; T, \sigma \rangle \rightarrow^* \langle T, \sigma'' \rangle$.

(4) Sean tt y ff los valores de verdad semánticos verdadero y falso, respectivamente:

Si $\sigma(B) = tt$, entonces $\langle$if B then S_1 else S_2 fi, $\sigma \rangle \rightarrow \langle S_1, \sigma \rangle$.

Si $\sigma(B) = ff$, entonces $\langle$if B then S_1 else S_2 fi, $\sigma\rangle \rightarrow \langle S_2, \sigma\rangle$.

En la instrucción de selección condicional, la evaluación de la guardia booleana B no modifica el estado corriente y se consume en un paso.
Notar que, como en el caso de $\sigma(e)$, la expresión $\sigma(B)$ es un abuso de notación.

(5) Si $\sigma(B) = ff$, entonces $\langle$while B do S od, $\sigma\rangle \rightarrow \langle E, \sigma\rangle$.
Si $\sigma(B) = tt$, entonces $\langle$while B do S od, $\sigma\rangle \rightarrow \langle S$; while B do S od, $\sigma\rangle$.

Como en la selección condicional, en la instrucción de repetición la evaluación de la guardia booleana B no modifica el estado corriente y se consume en un paso.

□

A partir de esta definición «por comprensión» de la semántica de PLW, utilizando inducción estructural, se puede desarrollar su definición «por extensión», enumerando las distintas formas de las computaciones de los programas de PLW. Para esto se formula el siguiente lema:

Lema de las formas de las computaciones de los programas de PLW. Dada la configuración inicial $C_0 = \langle S_0, \sigma_0\rangle$, las distintas formas de las computaciones $\pi(C_0)$ son las siguientes:

1) En los casos atómicos S_0 :: skip y S_0 :: x := e, la forma de $\pi(C_0)$ es como se estableció antes, porque tiene un solo paso.

2) En el caso $S_0 :: S_1 ; S_2$, $\pi(C_0)$ tiene una de las formas siguientes:

a) $\langle S_1 ; S_2, \sigma_0\rangle \rightarrow \langle T_1 ; S_2, \sigma_1\rangle \rightarrow \langle T_2 ; S_2, \sigma_2\rangle \rightarrow \ldots \rightarrow \langle T_k ; S_2, \sigma_k\rangle \rightarrow \ldots$, cuando S_1 no termina desde su estado inicial, por lo que $\pi(C_0)$ es infinita.

b) $\langle S_1 ; S_2, \sigma_0\rangle \rightarrow \ldots \rightarrow \langle S_2, \sigma_1\rangle \rightarrow \langle T_1, \sigma_2\rangle \rightarrow \langle T_2, \sigma_3\rangle \rightarrow \ldots \rightarrow \langle T_k, \sigma_{k+1}\rangle \rightarrow \ldots$, cuando S_1 termina desde su estado inicial y S_2 no termina desde su estado inicial, por lo que otra vez $\pi(C_0)$ es infinita.

c) $\langle S_1 ; S_2, \sigma_0\rangle \rightarrow \ldots \rightarrow \langle S_2, \sigma_1\rangle \rightarrow \ldots \rightarrow \langle E, \sigma_2\rangle$, cuando S_1 y S_2 terminan desde sus estados iniciales, por lo que $\pi(C_0)$ es finita.

3) En el caso S_0 :: if B then S_1 else S_2 fi , $\pi(C_0)$ tiene una de las formas siguientes:

a) <if B then S_1 else S_2 fi, σ_0> $\rightarrow$ <S_1, σ_0> $\rightarrow$ <T_1, σ_1> $\rightarrow$ <T_2, σ_2> $\rightarrow$ … $\rightarrow$ <T_k, σ_k> $\rightarrow$ …, cuando al comienzo vale B y S_1 no termina desde su estado inicial, por lo que $\pi(C_0)$ es infinita.

b) <if B then S_1 else S_2 fi, σ_0> $\rightarrow$ <S_1, σ_0> $\rightarrow$ … $\rightarrow$ <E, σ_1>, cuando al comienzo vale B y S_1 termina desde su estado inicial, por lo que $\pi(C_0)$ es finita.

(c) y (d) se definen de manera similar a (a) y (b), cuando B no se cumple al comienzo.

4) Por último, en el caso S_0 :: while B do S od , $\pi(C_0)$ tiene una de las formas siguientes:

a) <while B do S od, σ_0> $\rightarrow$ <E, σ_0>, cuando al comienzo no vale B, por lo que $\pi(C_0)$ es finita.

b) <while B do S od, σ_0> $\rightarrow$ … $\rightarrow$ <S ; while B do S od, σ_1> $\rightarrow$ <T_1 ; while B do S od, σ_2> $\rightarrow$ <T_2 ; while B do S od, σ_3> $\rightarrow$ … $\rightarrow$ <T_k ; while B do S od, σ_{k+1}> $\rightarrow$ …, cuando al comienzo vale B, y en alguna iteración S no termina desde su estado inicial, por lo que $\pi(C_0)$ es infinita.

c) <while B do S od, σ_0> $\rightarrow$ … $\rightarrow$ <while B do S od, σ_1> $\rightarrow$ … $\rightarrow$ <while B do S od, σ_2> $\rightarrow$ …, cuando toda vez que se evalúa la guardia booleana B ésta resulta verdadera, y en todas las iteraciones S termina desde su estado inicial, por lo que otra vez $\pi(C_0)$ es infinita.

d) <while B do S od, σ_0> $\rightarrow$ <S ; while B do S od, σ_0> $\rightarrow$ … $\rightarrow$ <E, σ_k>, cuando al comienzo vale B, en todas las iteraciones S termina desde su estado inicial, y alguna vez que se evalúa la guardia booleana B ésta resulta falsa, por lo que $\pi(C_0)$ es finita.

□

El ejemplo siguiente muestra la forma de la computación de un programa de PLW muy simple:

Ejemplo. Sea el programa S_{swap} :: z := x ; x := y ; y := z, y el estado inicial $\sigma_0 \models (x = 1 \wedge y = 2)$.

S_{swap} intercambia los contenidos de las variables x, y. La expresión $\sigma_0 \models (x = 1 \wedge y = 2)$ también se puede escribir $\sigma_0[x \mid 1][y \mid 2]$. Se cumple:

$$\begin{aligned}\pi(S_{swap}, \sigma_0) &= < z := x ; x := y ; y := z , \sigma_0[x \mid 1][y \mid 2] > \rightarrow < x := y ; y := z , \\ &\quad \sigma_0[x \mid 1][y \mid 2][z \mid x] > = \\ &= < x := y ; y := z , \sigma_0[x \mid 1][y \mid 2][z \mid 1] \rightarrow < y := z , \sigma_0[x \mid 1][y \\ &\quad \mid 2][z \mid 1][x \mid y] > = \\ &= < y := z , \sigma_0[x \mid 1][y \mid 2][z \mid 1][x \mid 2] > = < y := z , \sigma_0[y \mid 2][z \mid 1][x \\ &\quad \mid 2] > \rightarrow \\ &\rightarrow < E , \sigma_0[y \mid 2][z \mid 1][x \mid 2][y \mid z] > = < E , \sigma_0[y \mid 2][z \mid 1][x \mid 2][y \mid 1] > = \\ &= < E , \sigma_0[z \mid 1][x \mid 2][y \mid 1] >.\end{aligned}$$

Por lo tanto, se cumple $\models <x = 1 \wedge y = 2> S_{swap} <x = 2 \wedge y = 1>$.

De acuerdo a las definiciones previas, notar en el ejemplo que las expresiones del tipo $\sigma[x \mid a][x \mid b]$ se reemplazan por $\sigma[x \mid b]$ (la demostración de esta igualdad se pide en los ejercicios al final de esta parte del libro). También notar que las secuencias de la forma $S_1 ; S_2 ; S_3$ se interpretan como $S_1 ; (S_2 ; S_3)$. En realidad, se cumple la asociatividad del operador «**;**», por lo que se puede utilizar directamente la notación $S_1 ; S_2 ; \ldots ; S_n$

Ejercicio. *Probar que* $\mathcal{M}(S_1 ; (S_2 ; S_3)) = \mathcal{M}((S_1 ; S_2) ; S_3)$.

□

3.2.2. Método H para verificar la correctitud parcial de los programas de PLW

Con el método de verificación H se prueban, en el marco de los programas S ε PLW, las ternas de Hoare de correctitud parcial, es decir de la forma {p} S {q}, siendo p y q aserciones del lenguaje de especificación Assn (la lógica de primer orden con igualdad sobre el dominio de los números enteros).
Probar {p} S {q} significa construir una demostración de que la terna {p} S {q} es un teorema de H, lo que se denota con $\vdash_H \{p\}\ S\ \{q\}$, a partir de axiomas y reglas. Siendo H un método sensato (esto se probará después), se cumple que si vale $\vdash_H \{p\}\ S\ \{q\}$ entonces también vale

$\models$ {p} S {q}, es decir que el programa S es parcialmente correcto con respecto a la especificación <p, q>.
Primero se describen los axiomas y reglas de H, y se presentan ejemplos. Después se prueba la sensatez y la completitud del método.

3.2.2.1. Axiomas y reglas del método H

El método H tiene dos axiomas, cada uno asociado a una instrucción atómica de PLW, y cuatro reglas, tres asociadas a las instrucciones compuestas de secuencia, selección condicional y repetición, y la última de naturaleza especial, no relacionada con el lenguaje PLW:

(1) Axioma del skip (SKIP): para toda aserción p,

$$\{p\}\ \text{skip}\ \{p\}.$$

El axioma establece que si se cumple p antes de la ejecución de la instrucción skip, entonces sigue valiendo p después de la ejecución.

(2) Axioma de la asignación (ASI): para toda aserción p y toda asignación x := e de PLW,

$$\{p^x_e\}\ x := e\ \{p\}.$$

La expresión p^x_e denota la sustitución, en la aserción p, de todas las ocurrencias libres de la variable x por la expresión e.
El axioma establece que si se cumple p en términos de x después de la ejecución de x := e, entonces es que valía p en términos de e antes de la ejecución.

Ejemplo. Dada la postcondición $p = x \geq 0$ y la asignación x := x + 1, por el axioma ASI se cumple:

$$\{x + 1 \geq 0\}\ x := x + 1\ \{x \geq 0\}.$$

Es decir, si luego de la ejecución de x := x + 1 se cumple $x \geq 0$, esto significa que antes de la ejecución de la asignación valía $x + 1 \geq 0$.

Notar que el axioma ASI se lee «hacia atrás». La forma natural «hacia adelante» sería:

$\{true\}\ x := e\ \{x = e\}$,

pero esta fórmula no es verdadera, es decir, $\not\models \{true\}\ x := e\ \{x = e\}$. Por ejemplo, si $e = x + 1$, queda $\{true\}\ x := x + 1\ \{x = x + 1\}$, que no se cumple semánticamente. Esto se debe a que la x de la parte derecha de la postcondición se refiere a la variable antes de la asignación, mientras que la x de la parte izquierda de la postcondición se refiere a la variable luego de la asignación.

Una forma correcta del axioma ASI hacia adelante, justamente diferenciando las partes derecha e izquierda de la postcondición, sería:

$\{p\}\ x := e\ \{\exists z: p^x_z \wedge x = e^x_z\}$.

En lo que sigue, se utilizará la forma del axioma ASI hacia atrás. Es más simple, y además es la forma más difundida en la literatura. Esto producirá en el método H pruebas «hacia atrás», como se verá enseguida.

(3) Regla de la secuencia (SEC): dadas las aserciones p, q, r, y los subprogramas S_1 y S_2,

$$\frac{\{p\}\ S_1\ \{r\}\ ,\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$

La regla establece que si se cumple r cuando la ejecución de S_1 termina a partir de p, y si se cumple q cuando la ejecución de S_2 termina a partir de r, entonces se cumple q cuando la ejecución de la secuencia $S_1\ ;\ S_2$ termina a partir de p.

Para acortar las pruebas, se podrá utilizar directamente la siguiente generalización de la regla SEC, que se deriva de la forma original:

$$\frac{\{p\}\ S_1\ \{r_1\}\ ,\ \{r_1\}\ S_2\ \{r_2\}\ ,\ \{r_2\}\ S_3\ \{r_3\}\ ,\ \ldots\ ,\ \{r_{n-1}\}\ S_n\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ ;\ S_3\ ;\ \ldots\ ;\ S_n\ \{q\}}$$

Ejemplo. Volviendo al programa S_{swap} que intercambia los valores de las variables x, y, se prueba en H la fórmula:

$$\{x = X \wedge y = Y\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\},$$

empleando el axioma ASI repetidamente (una vez por cada asignación) y la regla SEC al final:

(1) $\{z = X \wedge x = Y\}\ y := z\ \{y = X \wedge x = Y\}$	(ASI)
(2) $\{z = X \wedge y = Y\}\ x := y\ \{z = X \wedge x = Y\}$	(ASI)
(3) $\{x = X \wedge y = Y\}\ z := x\ \{z = X \wedge y = Y\}$	(ASI)
(4) $\{x = X \wedge y = Y\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}$	(1, 2, 3, SEC)

Notar cómo el axioma ASI impone una forma de prueba hacia atrás, desde la postcondición hacia la precondición.

Se cumple, por la sensatez de H:

$$\models \{x = X \wedge y = Y\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}.$$

Si bien, obviamente, también se cumple:

$$\models \{y = Y \wedge x = X\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\},$$

notar, sin embargo, que el axioma ASI y la regla SEC no alcanzan para probar en H:

$$\vdash_H \{y = Y \wedge x = X\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}.$$

Esta incompletitud se corregirá enseguida, permitiendo en H manipular relaciones aritméticas. La idea es agregar a H el conjunto Tr de todas las aserciones verdaderas de la aritmética, más una regla de naturaleza especial para manipularlas, no relacionada con el lenguaje PLW, que se denomina regla de consecuencia (CONS).

La regla SEC refleja la propiedad de *composicionalidad* del método H (en realidad, de la programación secuencial en general). Si vale $\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$, también vale $\{p\}\ S_1\ ;\ S_2\ \{q\}$. La aserción intermedia r actúa como nexo y se ignora en la fórmula de correctitud resultante. Además, esto permite obtener programas equivalentes a $S_1\ ;\ S_2$ desde el punto de vista funcional, por ejemplo de la forma $S_1\ ;\ S_3$, si es que se cumple $\{r\}\ S_3\ \{q\}$.

(4) Regla de la selección condicional (COND): dadas las aserciones p y q, la expresión booleana B y los subprogramas S_1 y S_2,

$$\frac{\{p \wedge B\}\ S_1\ \{q\}\ ,\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi } \{q\}}$$

La regla establece que si se cumple q tanto cuando la ejecución de S_1 termina a partir de $(p \wedge B)$ como cuando la ejecución de S_2 termina a partir de $(p \wedge \neg B)$, entonces se cumple q cuando la ejecución de la selección condicional if B then S_1 else S_2 fi termina a partir de p.

Ejemplo. El siguiente programa implementa la especificación <x = X, y = |X|>, es decir, calcula en y el valor absoluto de x:

S_{va} :: if x > 0 then y := x else y := – x fi.

Se va a probar en H:

{true} if x > 0 then y := x else y := – x fi {y ≥ 0}.

Los primeros pasos de la prueba serían:

(1) {x ≥ 0} y := x {y ≥ 0} (ASI)
(2) {– x ≥ 0} y := – x {y ≥ 0} (ASI)

Para seguir, se debe poder aplicar la regla COND, y por lo tanto se necesita contar con un par de expresiones de la forma $(p \wedge B)$ y $(p \wedge \neg B)$. Asumiendo que en H se permite manipular relaciones aritméticas, la prueba entonces seguiría así:

(3) $\{true \wedge x > 0\}$ y := x {y ≥ 0} (1, MAT, CONS)
(4) $\{true \wedge \neg (x > 0)\}$ y := – x {y ≥ 0} (2, MAT, CONS)

MAT (por matemáticas) y CONS indican, respectivamente, que se utiliza una aserción verdadera del conjunto Tr, y que se aplica la regla de consecuencia. En el paso (3) se reemplaza la precondición de la fórmula de (1) por una aserción que la implica, y en el paso (4) se reemplaza la precondición de la fórmula de (2) por una aserción que la implica. Finalmente, empleando la regla COND se llega a:

(5) {true} if $x > 0$ then $y := x$ else $y := -x$ fi $\{y \geq 0\}$ (3, 4, COND)

La regla COND refleja que la selección condicional de PLW es una instrucción fácilmente verificable, con un único punto de entrada y un único punto de salida, a pesar de la bifurcación determinada por el valor de la guardia booleana B.

(5) Regla de la repetición (REP): dada la aserción p, la expresión booleana B y el subprograma S,

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \text{while B do S od}\ \{p \wedge \neg B\}}$$

La regla establece que si la ejecución de S, cuando termina, preserva p a partir de $(p \wedge B)$, entonces a partir de p, la ejecución de la repetición while B do S od cuando termina preserva p, cumpliéndose además $\neg B$. La aserción p es el *invariante* de la repetición: vale al comienzo, antes de cualquier iteración del cuerpo S, y al final si es que la repetición termina. Esto explica por qué la correctitud parcial se puede probar por inducción (este tipo de inducción se conoce como *computacional*). Esta noción es central en la actividad de programar: una repetición, en PLW como en cualquier lenguaje de programación, es la implementación de una propiedad invariante.
Después de la presentación de la siguiente regla, se presenta un ejemplo de aplicación de REP.

(6) Regla de consecuencia (CONS): dadas las aserciones p, q, p_1 y q_1, y el programa S,

$$\frac{p \rightarrow p_1\ ,\ \{p_1\}\ S\ \{q_1\}\ ,\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

La regla establece que si se cumple $\{p_1\}$ S $\{q_1\}$, entonces también se cumple cualquier fórmula de correctitud {p} S {q} tal que p sea más fuerte que p_1 (es decir, tal que implique p_1) y q sea más débil que q_1 (es decir, sea implicada por q_1). En particular, p y q pueden ser equivalentes a p_1 y q_1, respectivamente.

CONS es una regla de naturaleza especial. No se relaciona con el lenguaje de programación sino con el lenguaje de especificación. Dos de sus premisas no son fórmulas de correctitud sino aserciones.
La posibilidad de fortalecer precondiciones y debilitar postcondiciones permite «componer» ternas de Hoare. La figura 3.2.1 ilustra un ejemplo, en que a partir de $\{p\}\ S_1\ \{r_1\}$, $r_1 \rightarrow r_2$, y $\{r_2\}\ S_2\ \{q\}$, se obtiene $\{p\}\ S_1 ; S_2\ \{q\}$:

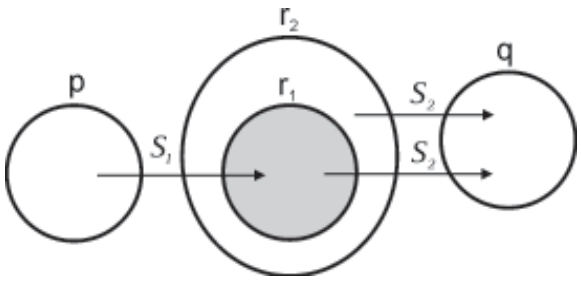


Figura 3.2.1

Por lo tanto, algunos pasos de una prueba de H pueden ser directamente aserciones (como se mencionó antes, se indican con el término MAT). Dado que el conjunto Tr de todas las aserciones verdaderas de la aritmética se agregan como premisas a H, la prueba de una fórmula de correctitud {p} S {q} en H se debe expresar en realidad de la siguiente manera:

$$Tr \vdash_H \{p\}\ S\ \{q\}.$$

Para simplificar la escritura, cuando quede claro por contexto se podrá utilizar directamente la notación $\vdash_H$, e incluso simplemente $\vdash$.

□

El siguiente ejemplo cierra la presentación del método H. Se muestra la utilización de las reglas REP y CONS:

Ejemplo. Sea otra vez el programa S_{fac} para calcular el factorial de un número. Se va a probar:

$\{x > 0\}\ S_{fac}$ :: a := 1 ; y := 1 ; while a < x do a := a + 1 ; y := y **.** a od $\{y = x!\}$,

y entonces, por la sensatez del método H, valdrá:

$\models \{x > 0\}\ S_{fac}\ \{y = x!\}$.

Se propone como invariante del while la aserción $p = (y = a! \wedge a \leq x)$. Para mayor claridad, la prueba se estructurará de la siguiente manera:

(a) $\vdash \{x > 0\}\ a := 1 ; y := 1\ \{y = a! \wedge a \leq x\}$.
(b) $\vdash \{y = a! \wedge a \leq x\}$ while $a < x$ do $a := a + 1 ; y := y \,.\, a$ od $\{y = x!\}$.
(c) $\vdash \{x > 0\}\ S_{fac}\ \{y = x!\}$, por la aplicación de la regla SEC sobre (a) y (b).

Prueba de (a):

(1) $\{1 = a! \wedge a \leq x\}\ y := 1\ \{y = a! \wedge a \leq x\}$ (ASI)
(2) $\{1 = 1! \wedge 1 \leq x\}\ a := 1\ \{1 = a! \wedge a \leq x\}$ (ASI)
(3) $\{1 = 1! \wedge 1 \leq x\}\ a := 1 ; y := 1\ \{y = a! \wedge a \leq x\}$ (1, 2, SEC)
(4) $x > 0 \rightarrow 1 = 1! \wedge 1 \leq x$ (MAT)
(5) $\{x > 0\}\ a := 1 ; y := 1\ \{y = a! \wedge a \leq x\}$ (3, 4, CONS)

Prueba de (b):

(6) $\{y \,.\, a = a! \wedge a \leq x\}\ y := y \,.\, a\ \{y = a! \wedge a \leq x\}$ (ASI)
(7) $\{y \,.\, (a + 1) = (a + 1)! \wedge (a + 1) \leq x\}\ a := a + 1\ \{y \,.\, a = a! \wedge a \leq x\}$ (ASI)
(8) $\{y \,.\, (a + 1) = (a + 1)! \wedge (a + 1) \leq x\}\ a := a + 1 ; y := y \,.\, a\ \{y = a! \wedge a \leq x\}$ (6, 7, SEC)
(9) $y = a! \wedge a \leq x \wedge a < x \rightarrow y \,.\, (a + 1) = (a + 1)! \wedge (a + 1) \leq x$ (MAT)
(10) $\{y = a! \wedge a \leq x \wedge a < x\}\ a := a + 1 ; y := y \,.\, a\ \{y = a! \wedge a \leq x\}$ (8, 9, CONS)
(11) $\{y = a! \wedge a \leq x\}$ while $a < x$ do $a := a + 1 ; y := y \,.\, a$ od $\{y = a! \wedge a \leq x \wedge \neg (a < x)\}$ (10, REP)
(12) $y = a! \wedge a \leq x \wedge \neg (a < x) \rightarrow y = x!$ (MAT)
(13) $\{y = a! \wedge a \leq x\}$ while $a < x$ do $a := a + 1 ; y := y \,.\, a$ od $\{y = x!\}$ (11, 12, CONS)

Prueba de (c):

(14) $\{x > 0\}\ S_{fac}\ \{y = x!\}$ (5, 13, SEC)

Como se observa en el ejemplo, aún para un programa sencillo la prueba puede ser muy larga. En los ejemplos siguientes se agruparán varios pasos en uno.

Por otra parte, al final de este capítulo se va a describir una manera alternativa de presentar una prueba, la *proof outline* (esquema de prueba), que no es más que un programa anotado adecuadamente, con los pasos de la prueba intercalados entre sus instrucciones. Así, las demostraciones son mucho más estructuradas, y además los programas quedan documentados de un modo apropiado.

3.2.2.2. Sensatez del método H

En esta sección se demuestra que el método H es sensato, es decir que todas las fórmulas de correctitud parcial {p} S {q} probadas en H son verdaderas.

Teorema. Para todo par de aserciones p y q, y para todo programa S ε PLW, se cumple:

$$Tr \vdash_H \{p\}\ S\ \{q\} \quad \rightarrow \quad \models \{p\}\ S\ \{q\}.$$

Prueba. Para demostrar que el método H es sensato, se probará que los axiomas son verdaderos y que las reglas son sensatas, es decir que producen conclusiones verdaderas a partir de premisas verdaderas. Se va a utilizar inducción sobre la longitud de la prueba.

(1) El axioma SKIP es verdadero.

Hay que probar que para toda aserción p: $\models \{p\}$ skip $\{p\}$.
Sea $\sigma_0 \models p$. Según la semántica de PLW: $\langle skip, \sigma_0\rangle \rightarrow \langle E, \sigma_0\rangle$. Entonces el estado final σ_0 cumple $\sigma_0 \models p$.
Por lo tanto, $\models \{p\}$ skip $\{p\}$.

(2) El axioma ASI es verdadero.

Hay que probar que para toda aserción p y para toda asignación x := e de PLW: $\models \{p^x_e\}\ x := e\ \{p\}$.
Sea $\sigma_0 \models p^x_e$. Según la semántica de PLW: $\langle x := e, \sigma_0\rangle \rightarrow \langle E, \sigma_0[x \mid e]\rangle$. Entonces el estado final $\sigma_0[x \mid e]$ cumple $\sigma_0[x \mid e] \models p$.
Por lo tanto, $\models \{p^x_e\}\ x := e\ \{p\}$.

(3) La regla SEC es sensata.

Sea $\vdash \{p\}\ S_1 ; S_2\ \{q\}$. Hay que probar $\models \{p\}\ S_1 ; S_2\ \{q\}$.
Si $\vdash \{p\}\ S_1 ; S_2\ \{q\}$, entonces antes tuvo que suceder: $\vdash \{p\}\ S_1\ \{r\}$ y $\vdash \{r\}\ S_2\ \{q\}$, dada una determinada aserción r, para poder aplicar la regla SEC (se obvia en este caso y los siguientes la aplicación de la regla CONS, cuya sensatez se prueba al final). Estas pruebas son más cortas que la prueba de $\{p\}\ S_1 ; S_2\ \{q\}$, por lo que aplicando la hipótesis inductiva: $\models \{p\}\ S_1\ \{r\}$ y $\models \{r\}\ S_2\ \{q\}$.
Sea $\sigma_0 \models p$, y supóngase que $val(\pi(S_1 ; S_2, \sigma_0)) \neq \perp$ (si $val(\pi(S_1 ; S_2, \sigma_0)) = \perp$ no se viola $\models \{p\}\ S_1 ; S_2\ \{q\}$).
Según la semántica de PLW: $\langle S_1 ; S_2, \sigma_0\rangle \rightarrow^* \langle S_2, \sigma_1\rangle \rightarrow^* \langle E, \sigma_2\rangle$.
Como $\models \{p\}\ S_1\ \{r\}$ entonces $\sigma_1 \models r$, y como $\models \{r\}\ S_2\ \{q\}$ entonces $\sigma_2 \models q$.

Por lo tanto, $\models \{p\}\ S_1 ; S_2\ \{q\}$.

(4) La regla COND es sensata.

Sea $\vdash \{p\}$ if B then S_1 else S_2 fi $\{q\}$. Hay que probar $\models \{p\}$ if B then S_1 else S_2 fi $\{q\}$.
Si $\vdash \{p\}$ if B then S_1 else S_2 fi $\{q\}$, entonces antes tuvo que suceder: $\vdash \{p \wedge B\}\ S_1\ \{q\}$ y $\vdash \{p \wedge \neg B\}\ S_2\ \{q\}$, para poder aplicar la regla COND. Estas pruebas son más cortas que la prueba de $\{p\}$ if B then S_1 else S_2 fi $\{q\}$, por lo que aplicando la hipótesis inductiva: $\models \{p \wedge B\}\ S_1\ \{q\}$ y $\models \{p \wedge \neg B\}\ S_2\ \{q\}$.
Sea $\sigma_0 \models p$, y supóngase que $val(\pi($if B then S_1 else S_2 fi, $\sigma_0)) \neq \perp$.
Según la semántica de PLW:
$\langle$if B then S_1 else S_2 fi, $\sigma_0\rangle \rightarrow \langle S_1, \sigma_0\rangle \rightarrow^* \langle E, \sigma_1\rangle$ si $\sigma_0(B) =$ tt, o bien
$\langle$if B then S_1 else S_2 fi, $\sigma_0\rangle \rightarrow \langle S_2, \sigma_0\rangle \rightarrow^* \langle E, \sigma_2\rangle$ si $\sigma_0(B) =$ ff.
Como $\models \{p \wedge B\}\ S_1\ \{q\}$ y $\models \{p \wedge \neg B\}\ S_2\ \{q\}$, entonces $\sigma_1 \models q$ y también $\sigma_2 \models q$.

Por lo tanto, $\models \{p\}$ if B then S_1 else S_2 fi $\{q\}$.

(5) La regla REP es sensata.

Sea $\vdash \{p\}$ while B do S od $\{p \wedge \neg B\}$. Hay que probar $\models \{p\}$ while B do S od $\{p \wedge \neg B\}$.

Si |– {p} while B do S od {p $\wedge \neg$ B}, entonces antes tuvo que suceder: |– {p $\wedge$ B} S {p}, para poder aplicar la regla REP. Esta prueba es más corta que la prueba de {p} while B do S od {p $\wedge \neg$ B}, por lo que aplicando la hipótesis inductiva: |= {p $\wedge$ B} S {p}.
Sea σ_0 |= p y supóngase que val(π(while B do S od, σ_0)) $\neq \perp$.
Según la semántica de PLW:
$C_0 \rightarrow^* C_1 \rightarrow^* C_2 \rightarrow^* \ldots \rightarrow^* C_{n-1} \rightarrow^* C_n$, tal que n > 0, C_i = <while B do S od, σ_i> con 0 $\leq$ i < n, y C_n = <E, σ_n>, cumpliéndose σ_n(B) = ff. Como |= {p $\wedge$ B} S {p}, entonces σ_n |= p $\wedge \neg$ B.

Por lo tanto, |= {p} while B do S od {p $\wedge \neg$ B}.

(6) La regla CONS es sensata.

Sea |– {p} S {q}, habiéndose aplicado antes la regla CONS. Hay que probar |= {p} S {q}.
Si |– {p} S {q} y se aplicó CONS, entonces antes tuvo que suceder en el caso más general:
|– {p_1} S {q_1}, p $\rightarrow$ p_1 y q_1 $\rightarrow$ q. La prueba de {p_1} S {q_1} es más corta que la de {p} S {q}, por lo que aplicando la hipótesis inductiva: |= {p_1} S {q_1}.
Sea σ_0 |= p y supóngase que val(π(S, σ_0)) $\neq \perp$, es decir: <S, σ_0> $\rightarrow^*$ <E, σ_1>.
Como p $\rightarrow$ p_1, entonces σ_0 |= p_1.
Como |= {p_1} S {q_1}, entonces σ_1 |= q_1.
Finalmente, como q_1 $\rightarrow$ q, entonces σ_1 |= q.

Por lo tanto, |= {p} S {q}.

□

Antes de pasar a la prueba de completitud del método H, cabe remarcar que ya agregando a PLW una estructura de datos tan simple como los arreglos de una dimensión, se puede impactar la sensatez de H. De hecho, como se muestra en el siguiente ejemplo, el axioma ASI planteado deja de valer:

Ejemplo. Sea el siguiente esquema de fórmula de correctitud:

{p} a[a[2]] := 3 {a[a[2]] = 3}.

Por ASI, $p = (a[a[2]] = 3)^{a[a[2]]}_{3}$ = true, y así, por la sensatez de H, queda la fórmula:

$\models \{true\}\ a[a[2]] := 3\ \{a[a[2]] = 3\}$.

Pero $\not\models \{true\}\ a[a[2]] := 3\ \{a[a[2]] = 3\}$: si $\sigma_0(a[2]) = 2$ y $\sigma_0(a[3]) = 2$, por la semántica de PLW queda: $<a[a[2]] := 3, \sigma_0> \rightarrow <E, \sigma_1>$, con $\sigma_1(a[2]) = 3$ y $\sigma_1(a[3]) = 2$, y por lo tanto $\sigma_1(a[a[2]]) = 2$.

Por otro lado, el axioma ASI no es verdadero aún si no se permiten expresiones en índices que tengan variables indexadas. Sea el siguiente esquema de fórmula de correctitud:

$\{p\}\ a[x] := 0\ \{a[x] + 1 = a[y]\}$.

Por ASI, $p = (a[x] + 1 = a[y])^{a[x]}_{0} = (0 + 1 = a[y])$, y así, por la sensatez de H, queda:

$\models \{0 + 1 = a[y]\}\ a[x] := 0\ \{a[x] + 1 = a[y]\}$.

Pero $\not\models \{0 + 1 = a[y]\}\ a[x] := 0\ \{a[x] + 1 = a[y]\}$: si $\sigma_0(a[1]) = 1$, $\sigma_0(x) = 1$ y $\sigma_0(y) = 1$, por la semántica de PLW queda: $<a[x] := 0, \sigma_0> \rightarrow <E, \sigma_1>$, con $\sigma_1(a[x]) = 0$ y $\sigma_1(a[y]) = 0$, y por lo tanto $\sigma_1(a[x] + 1) = 1 \neq \sigma_1(a[y])$.

Hay básicamente dos aproximaciones para que el axioma ASI vuelva a valer en este caso. No se van a desarrollar porque su tratamiento está fuera del alcance de este libro. Sólo se van a comentar a continuación sus principales características:
Por un lado, la aproximación más familiar se basa en asignaciones a componentes, y consiste en definir adecuadamente la sustitución sintáctica $p^{a[e]}_{e'}$, dado el arreglo a y las expresiones enteras e, e', para que se cumpla: $(\sigma[a[e] \mid e'] \models p) \leftrightarrow (\sigma \models p^{a[e]}_{e'})$.
La extensión semántica de PLW para el uso de arreglos asociada a esta primera aproximación, asumiendo para simplificar la presentación que las expresiones de los índices no tienen variables indexadas, es:

- Se particiona el conjunto de variables enteras Ivar de la forma: Ivar = Svar $\cup$ Avar, donde Svar es el conjunto de las variables simples y Avar es el conjunto de los arreglos.

- Los estados $\sigma \varepsilon \Sigma$ asignan enteros de V a los elementos $x \varepsilon$ Svar y a los elementos $a[i] \varepsilon$ Avar. Los elementos a[i] se denotan con <a, i>, y se define $\sigma(a[e]) = \sigma(<a, \sigma(e)>)$.
- La variante de un estado, considerando una variable indexada es: $\sigma[a[e] \mid e'](<b, i>) = \sigma(e')$, si a = b y $\sigma(e) = i$, y $\sigma[a[e] \mid e'](<b, i>) = \sigma(<b, i>)$, en caso contrario.
- La relación «$\rightarrow$» definida para la instrucción de asignación a una variable indexada es: $<a[e] := e', \sigma> \rightarrow <E, \sigma[a[e] \mid e']>$.

La segunda aproximación, menos familiar, se basa en asignaciones totales. Es decir, la asignación a[e] := e' se interpreta como a := f(a, e, e'), que es una asignación al arreglo completo a, no sólo a uno de sus componentes a[i].
En este caso, la semántica de PLW contempla estados $\sigma \varepsilon \Sigma$ que asignan valores de tipo arreglo de enteros a variables de tipo arreglo de enteros, y la sustitución sintáctica que hay que definir adecuadamente tiene la forma $p^a_{f(a, e, e')}$.
Notar que en ninguna de las dos aproximaciones se ha considerado la posibilidad de violación de rango de un arreglo. Si se la considerara, debería introducirse en la semántica un estado de falla, por ejemplo f, para indicar terminación incorrecta. Como en el caso de $\perp$, valdría:

- Para toda aserción p, $f \not\models p$.
- Si $val(\pi(S_1, \sigma)) = f$, entonces $val(\pi(S_1 ; S_2, \sigma)) = f$ (es decir, la falla se propaga).

En realidad, el estado f ya podía haberse introducido en las asignaciones x := e, contemplando subexpresiones de e de la forma z / 0, z mod 0, etc. De todos modos, se ha optado por introducir este estado especial más adelante. En el marco del tipo de programas tratados en este capítulo, se asume que las asignaciones terminan siempre correctamente, por lo que tienen que haber convenciones del tipo $\sigma(z / 0) = 0$, $\sigma(z \bmod 0) = \sigma(z)$, etc.

3.2.2.3. Completitud del método H

En esta sección se trata la demostración de que H es completo, es decir, que todas las fórmulas de correctitud parcial {p} S {q} verdaderas pueden ser probadas en H.

Ante todo, debe decirse que la completitud de H no puede ser absoluta, en los siguientes dos sentidos:
No todas las aserciones del conjunto Tr se pueden probar, de acuerdo al Teorema de Incompletitud de Gödel. De esta manera, si por ejemplo se cumple $\models$ {true} skip {p}, tal que p ε Tr es una de dichas aserciones, no podrá probarse dicha terna en H, porque debería probarse la implicación true $\rightarrow$ p, es decir p. En la práctica, de todos modos, esto no tiene relevancia alguna, porque de lo que se trata es de probar programas, no propiedades de los números. Se asumirá que Tr está incluido en el conjunto de axiomas de H.
Aún ignorando el caso anterior, la completitud de H depende de si el lenguaje de aserciones Assn es expresivo, es decir, si con Assn se pueden describir, en todos los casos, los conjuntos de estados intermedios considerados en las pruebas de correctitud. La expresividad de Assn depende entre otras cosas, de la riqueza de operaciones aritméticas con que esté dotado el conjunto de expresiones enteras. Se asumirá que Assn es expresivo. En particular, se asumirá que para toda aserción p y para todo programa S, puede expresarse en Assn el conjunto de estados post(p, S), que se define de la siguiente manera:

$$post(p, S) = \{\sigma' \mid \exists\sigma: \sigma \models p \wedge \mathcal{M}(S)(\sigma) = \sigma' \neq \perp\}.$$

Es decir, post(p, S) es el conjunto de todos los estados finales, que se obtienen de ejecutar el programa S a partir de todos los estados iniciales que satisfacen la precondición p. O en otras palabras, post(p, S) es la postcondición más fuerte dados la precondición p y el programa S.
Con las consideraciones anteriores, ahora sí se va a plantear y demostrar el teorema de completitud del método H. Se dirá que la completitud de H es *relativa*.

Teorema. Para todo par de aserciones p y q, y para todo programa S ε PLW, se cumple:

$$\models \{p\}\ S\ \{q\} \quad \rightarrow \quad Tr \vdash_H \{p\}\ S\ \{q\}.$$

Prueba. Para demostrar que el método H es (relativamente) completo, se probará que cualquiera sea la forma del programa S ε PLW (skip, asignación, secuencia, selección condicional o repetición), si se cumple $\models$ {p} S {q}, entonces se puede probar {p} S {q} en H. Se va a utilizar inducción sobre la estructura de S.

(1) Caso de S :: skip.

Dado $\models \{p\}$ skip $\{q\}$, hay que probar Tr $\vdash_H \{p\}$ skip $\{q\}$.
Por la semántica de PLW: $\langle skip, \sigma\rangle \rightarrow \langle E, \sigma\rangle$. Si $\sigma \models p$, entonces $\sigma \models q$, y por lo tanto $p \rightarrow q$. La siguiente es una prueba de $\{p\}$ skip $\{q\}$ en H:
a) $\{q\}$ skip $\{q\}$, por (SKIP).
b) $p \rightarrow q$, por (MAT).
c) $\{p\}$ skip $\{q\}$ por (a, b, CONS).

Por lo tanto, $\vdash \{p\}$ skip $\{q\}$.

(2) Caso de S :: x := e.

Dado $\models \{p\}$ x := e $\{q\}$, hay que probar Tr $\vdash_H \{p\}$ x := e $\{q\}$.
Por la semántica de PLW: $\langle x := e, \sigma\rangle \rightarrow \langle E, \sigma[x \mid e]\rangle$. Si $\sigma \models p$, entonces $\sigma[x \mid e] \models q$, es decir $\sigma \models q^x_e$, y por lo tanto $p \rightarrow q^x_e$. La siguiente es una prueba de $\{p\}$ x := e $\{q\}$ en H:
a) $\{q^x_e\}$ x := e $\{q\}$, por (ASI).
b) $p \rightarrow q^x_e$, por (MAT).
c) $\{p\}$ x := e $\{q\}$, por (a, b, CONS).

Por lo tanto, $\vdash \{p\}$ x := e $\{q\}$.

(3) Caso de S :: S_1 ; S_2.

Dado $\models \{p\}\ S_1 ; S_2\ \{q\}$, hay que probar Tr $\vdash_H \{p\}\ S_1 ; S_2\ \{q\}$.
Sea r = post(p, S_1). La aserción r existe por la expresividad de Assn. Así, por la definición de post(p, S_1) y la semántica de PLW, se cumple: $\models \{p\}\ S_1 \{r\}$, y $\models \{r\}\ S_2\ \{q\}$.
Aplicando la hipótesis inductiva, dado que S_1 y S_2 son subprogramas de S, queda: $\vdash \{p\}\ S_1 \{r\}$, y $\vdash \{r\}\ S_2\ \{q\}$.

Finalmente, aplicando SEC: $\vdash \{p\}\ S_1 ; S_2\ \{q\}$.

(4) Caso de S :: if B then S_1 else S_2 fi.

Dado $\models \{p\}$ if B then S_1 else S_2 fi $\{q\}$, hay que probar Tr $\vdash_H \{p\}$ if B then S_1 else S_2 fi $\{q\}$.

Por la semántica de PLW, se cumple: $\models \{p \wedge B\}\ S_1 \{q\}$, y $\models \{p \wedge \neg B\}\ S_2 \{q\}$.

Aplicando la hipótesis inductiva, dado que S_1 y S_2 son subprogramas de S, queda: $\vdash \{p \wedge B\}\ S_1 \{q\}$, y $\vdash \{p \wedge \neg B\}\ S_2 \{q\}$.
Finalmente, aplicando COND: $\vdash \{p\}$ if B then S_1 else S_2 fi $\{q\}$.

(5) Caso de S :: while B do S_1 od.

Dado $\models \{r\}$ while B do S_1 od $\{q\}$, hay que probar Tr $\vdash_H \{r\}$ while B do S_1 od $\{q\}$.

Por la semántica de PLW, hay que encontrar un invariante p que cumpla: $r \rightarrow p$, $\models \{p \wedge B\}\ S_1 \{p\}$, y $p \wedge \neg B \rightarrow q$. El invariante p debe denotar el conjunto de todos los estados obtenidos, a partir de la precondición r, a lo largo de las n iteraciones de S_1 (que pueden ser infinitas). Es decir:

$p = p_0 \vee p_1 \vee \ldots \vee p_k \vee \ldots$, con $p_0 = r$, y $p_{i+1} = post(p_i \wedge B, S_1)$ para todo $i \geq 0$.

Las aserciones $post(p_i \wedge B, S_1)$ existen por la expresividad de Assn. Por el mismo motivo se puede probar (no se va a desarrollar) que la disyunción infinita $\vee_i p_i$ es expresable en Assn mediante una aserción finita p.
Aplicando la hipótesis inductiva, dado que S_1 es un subprograma de S, queda: $\vdash \{p \wedge B\}\ S_1 \{p\}$.
Finalmente, aplicando REP y CONS: $\vdash \{r\}$ while B do S_1 od $\{q\}$.

□

3.2.3. Método H* para verificar la correctitud total de los programas de PLW

Con el método de verificación H* se prueban, en el marco de los programas S ε PLW, las ternas de Hoare de correctitud total (correctitud parcial más terminación), es decir de la forma <p> S <q>, siendo p y q aserciones del lenguaje de especificación Assn.
La idea es demostrar por medio de axiomas y reglas, que <p> S <q> es un teorema de H*, lo que se denota con Tr $\vdash_{H^*}$ <p> S <q>.

H* es un método sensato. Por lo tanto, probando <p> S <q> en H*, se prueba $\models$ <p> S <q>, es decir que el programa S es totalmente correcto con respecto a la especificación <p, q>. También se cumple que H* es relativamente completo.
Los axiomas y reglas de H* son los de H, salvo la regla de la repetición, dado que en PLW la instrucción while es la única que puede provocar divergencia. En todos los casos, los nombres de los axiomas y reglas de H* son los de H, pero para distinguirlos se les agrega al final el símbolo «*****», y además las pre y postcondiciones se delimitan por «< >» en lugar de «{ }». Por ejemplo, el axioma SKIP* tiene la forma <p> skip <p>.
La regla de la repetición (REP*) del método H* se formula de la siguiente manera: dada la aserción p, la expresión booleana B, el subprograma S, y una variable n natural que no forma parte de las variables de S,

$$\frac{p(n+1) \rightarrow B \ , \ <p(n+1)> \ S \ <p(n)> \ , \ p(0) \rightarrow \neg B}{<\exists n{:}p(n)> \text{ while } B \text{ do } S \text{ od } <p(0)>}$$

Como se aprecia, se mantiene el concepto de invariante p, que la ejecución de S preserva cuando se ejecuta a partir de la condición ($p \wedge B$). Por eso, tal como lo establece la regla, p sigue valiendo cuando la instrucción de repetición termina.
Ahora además, se le agrega a p un parámetro (variable n) que varía en el dominio de los números naturales, que no forma parte del conjunto de variables del subprograma S. La variable n debe interpretarse como una variable de especificación, y por lo tanto está implícitamente cuantificada universalmente.
En lo que tiene que ver con el parámetro n, la regla REP* establece que si:

- mientras $n \neq 0$, la guardia booleana B es verdadera,
- S decrementa en uno el valor de n toda vez que se ejecuta, y
- cuando n = 0, la guardia booleana B es falsa,

entonces se cumple que existe un número natural n, a partir del cual la instrucción while B do S od termina, y a la finalización el valor de n es cero. Es decir que n representa la cantidad de iteraciones que ejecuta el while. Al terminar la repetición, se cumple la condición $\neg B$, porque $p(0) \rightarrow \neg B$.

El invariante parametrizado p(n) es una abreviatura de: $p \wedge n \varepsilon N \wedge n = t$, siendo t una expresión entera definida en términos de las variables

del subprograma S. A la expresión t se la conoce como *función cota* (*bound function*).
Se podría formular una regla alternativa más flexible, que no exija estrictamente un decremento en uno de la variable n, ni que al final el valor de n sea el cero.
Por lo tanto, a diferencia de la correctitud parcial, la terminación no se prueba por inducción. En efecto, se debe encontrar un *variante* n que represente progreso hacia la terminación.

□

En el ejemplo siguiente se presenta una prueba dentro del método H*:

Ejemplo. Ya se verificó con el método H, la correctitud parcial del programa S_{fac} con respecto a la especificación <x > 0, y = x!>, es decir:

{x > 0} S_{fac} :: a := 1 ; y := 1 ; while a < x do a := a + 1 ; y := y **.** a od {y = x!}.

Ahora se pretende verificar con el método H*, la correctitud total:

<x >0> S_{fac} <y = x!>.

Por el Lema de Separación, bastará entonces con probar en H*:

<x >0> S_{fac} <true>.

Se propone como invariante parametrizado de la repetición, la aserción p(n) = (n = x – a), que en realidad abrevia: true $\wedge$ n ε N $\wedge$ n = x – a.

Para acortar la prueba, en algunos casos se van a agrupar varios pasos en uno. Además, para mayor claridad, la prueba se estructurará de la siguiente manera:

(a) |– <x > 0> a := 1 ; y := 1 <$\exists$n: n = x – a>.
(b) |– <$\exists$n: n = x – a> while a < x do a := a + 1 ; y := y **.** a od <true>.
(c) |– <x > 0> S_{fac} <true>, por aplicación de SEC sobre (a) y (b).

Prueba de (a):

(1) <$\exists$n: n = x – 1> a := 1; y := 1 <$\exists$n: n = x – a>	(ASI*, SEC*)
(2) x > 0 $\rightarrow$ $\exists$n: n = x – 1	(MAT)
(3) <x > 0> a := 1; y := 1 <$\exists$n: n = x – a>	(1, 2, CONS*)

Prueba de (b):

(4) $n + 1 = x - a \rightarrow a < x$ (MAT)

(5) $0 = x - a \rightarrow \neg (a < x)$ (MAT)

(6) $<n + 1 = x - a>$ a := a + 1 ; y := y . a $<n = x - a>$ (ASI*, SEC*,CONS*)

(7) $<\exists n: n = x - a>$ while a < x do a := a + 1 ; y := y . a od $<0 = x - a>$ (4, 5, 6, REP*)

(8) $0 = x - a \rightarrow$ true (MAT)

(9) $<\exists n: n = x - a>$ while a < x do a := a + 1 ; y := y . a od <true> (7, 8, CONS*)

Prueba de (c):

(10) $<x > 0> S_{fac}$ <true> (3, 9, SEC*)

Se cumple que H* es sensato, es decir que todas las fórmulas de correctitud total <p> S <q> probadas en H* son verdaderas, lo que se denota de la siguiente manera:

Tr $|-_{H^*}$ <p> S <q> $\rightarrow$ $|=$ <p> S <q>, para todo par <p, q> y para todo S.

La prueba de que H* – {REP*} es sensato ya se hizo cuando se demostró la sensatez del método H. Por lo tanto, sólo faltaría probar que:

Tr $|-_{H^*}$ $<\exists n: p(n)>$ while B do S od $<p(0)> \rightarrow |=$ $<\exists n: p(n)>$ while B do S od <p(0)>.

Se utiliza inducción sobre la longitud de la prueba. Si Tr $|-_{H^*}$ $<\exists n: p(n)>$ while B do S od <p(0)>, entonces tuvo que haberse probado antes en H*:

(a) $p(n + 1) \rightarrow$ B, (b) $<p(n + 1)>$ S $<p(n)>$, y (c) $p(0) \rightarrow \neg$ B.

Por la hipótesis inductiva, se cumple:

(d) $|=$ $<p(n + 1)>$ S $<p(n)>$.

Sea $\sigma |=$ $\exists n: p(n)$, y supóngase que la repetición no termina. Si n = 0, entonces por (c) y la semántica de PLW, la repetición termina (absurdo). Y si $n \neq 0$, entonces por (a), (d) y la semántica de PLW, debe existir una cadena descendente infinita en (N, <) (absurdo).

□

También se cumple que H* es completo, es decir que todas las fórmulas de correctitud total <p> S <q> verdaderas pueden ser probadas en H*, lo que se denota de la siguiente manera:

$\models$ <p> S <q> $\rightarrow$ Tr $\vdash_{H^*}$ <p> S <q>, para todo par <p, q> y para todo S.

En realidad, el método H* es relativamente completo, teniendo en cuenta las mismas observaciones efectuadas en la prueba de completitud de H, pero ahora se requiere, además, que el variante n = t, siendo n ε N y t una expresión entera, sea expresable en el lenguaje de aserciones Assn.

3.2.4. Otras consideraciones

A los efectos de acortar las pruebas en los métodos H y H*, se pueden agregar reglas de la misma manera que se hace en cualquier sistema deductivo. Las reglas son redundantes pero simplifican pasos.
Dos ejemplos en el método H son:

Regla de la conjunción (AND): dadas las aserciones p, q y r, y el programa S,

$$\frac{\{p\}\ S\ \{q\}\ ,\ \{p\}\ S\ \{r\}}{\{p\}\ S\ \{q \wedge r\}}$$

La regla establece que si después de la ejecución del programa S a partir de la precondición p se cumple tanto la postcondición q como la postcondición r, entonces después de la ejecución de S a partir de p se cumple q $\wedge$ r.

Regla de la disyunción (OR): dadas las aserciones p, q y r, y el programa S,

$$\frac{\{p\}\ S\ \{q\}\ ,\ \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$$

La regla establece que si después de la ejecución del programa S tanto a partir de la precondición p como de la precondición r se cumple la

postcondición q, entonces después de la ejecución de S a partir de p $\vee$ r se cumple q.

Ejercicio. *Probar que H sigue siendo un método sensato si se le agregan las reglas AND y OR.*

□

Hay una presentación alternativa de prueba en H y H*, la proof outline, que consiste en intercalar los pasos de la prueba entre las instrucciones del programa. El objetivo es obtener una demostración mucho más estructurada, que además documenta adecuadamente el programa. Como se verá después, en la verificación de los programas concurrentes las proof outlines son imprescindibles.

Definición. Dadas las aserciones p y q ε Assn y el programa S ε PLW, una *proof outline de correctitud parcial* de {p} S {q} es una anotación de S, denotada con S* (o también con {p} S* {q} cuando es útil explicitar la pre y postcondición), que se compone por aserciones intercaladas antes y después de los subprogramas de S. La definición inductiva de S* es la siguiente:

(1) $$\{p\}\ \text{skip}\ \{p\}.$$

(2) $$(p^x_e\}\ x := e\ \{p\}.$$

(3) $$\frac{\{p\}\ S_1^*\ \{r\}\ ,\ \{r\}\ S_2^*\ \{q\}}{\{p\}\ S_1^*\ \{r\}\ ;\ S_2^*\ \{q\}}$$

(4) $$\frac{\{p \wedge B\}\ S_1^*\ \{q\}\ ,\ \{p \wedge \neg B\}\ S_2^*\ \{q\}}{\{p\}\ \text{if B then } \{p \wedge B\}\ S_1^*\ \{q\}\ \text{else } \{p \wedge \neg B\}\ S_2^*\ \{q\}\ \text{fi } \{q\}}$$

(5) $$\frac{\{p \wedge B\}\ S^*\ \{p\}}{\{\text{inv: } p\}\ \text{while B do } \{p \wedge B\}\ S^*\ \{p\}\ \text{od } \{p \wedge \neg B\}}$$

(6) $$\frac{p \rightarrow p_1\ ,\ \{p_1\}\ S^*\ \{q_1\}\ ,\ q_1 \rightarrow q}{\{p\}\{p_1\}\ S^*\ \{q_1\}\{q\}}$$

(7) $$\frac{\{p\}\ S^*\ \{q\}}{\{p\}\ S^{**}\ \{q\}},$$

tal que en S** se pueden omitir algunas aserciones de S*, entre las que no pueden estar los invariantes de los while.

□

Una proof outline de correctitud parcial de {p} S {q} (o una proof outline de correctitud parcial {p} S* {q}) es *estándar*, si a todo subprograma T de S lo precede exactamente una aserción de S*, referida como pre(T), y le sigue exactamente una aserción de S*, referida como post(T).
Las *proof outlines de correctitud total* <p> S* <q> se definen como las proof outlines de correctitud parcial, salvo que utilizan los símbolos «< >» en lugar de los símbolos «{ }», e incluyen antes de cada while no sólo el invariante sino también el variante.
Se presenta a continuación un ejemplo de proof outline:

Ejemplo. La prueba de correctitud total desarrollada previamente del programa S_{fac} que calcula el factorial de un número, puede ser mostrada alternativamente en forma de proof outline de la siguiente manera:

```
        <x > 0>
S_fac ::  a := 1 ; y := 1 ;
        <inv: y = a! ∧ a ≤ x , fc: x – a>
          while a < x do
        <y = a! ∧ a < x>
          a := a + 1 ; y := y . a
        <y = a! ∧ a ≤ x>
                      od
        <y = x!>
```

En la proof outline del ejemplo se han omitido algunas aserciones (no es estándar). Antes del while se especifica el invariante, precedido por «inv:», y el variante, precedido por «fc:» (por función cota). Notar que no se anota la prueba de terminación, sino que sólo se muestra el variante.
Se prueba fácilmente por inducción el Lema de Preservación Composicional, que establece que, dada una proof outline de correctitud parcial estándar {p} S* {q} y un subprograma T de S, en toda ejecución de S a partir de un estado inicial que satisface p, se cumple que si T se ejecuta en un estado σ y termina en un estado σ', entonces $\sigma \models$ pre(T)

y $\sigma' \models$ post(T). Esta propiedad del método H se conoce como *sensatez fuerte* (*strong soundness*).

Ejercicio. *Probar el Lema de Preservación Composicional.*

□

El concepto de proof outline <p> S* <q> constituye una guía para el desarrollo sistemático del programa S con respecto a <p, q>.
Por ejemplo, supóngase que se pretende construir un típico programa S con un conjunto de inicializaciones seguido de una repetición, es decir:

$S :: S_1$; while B do S_2 od,

tal que S satisfaga la especificación <r, q>.

La construcción de S consistirá fundamentalmente en la búsqueda y posterior implementación en términos de la instrucción while, de un invariante p y un variante decreciente n. Considerando el esquema más flexible en que no se requiere un decremento estricto en uno del variante, ni que éste llegue a cero, entonces se debería cumplir:

a) $<r> S_1 <p>$.
b) $<p \wedge B> S_2 <p>$.
c) $p \wedge \neg B \rightarrow q$.
d) $p \rightarrow n \geq 0$.
e) $<p \wedge B \wedge n = t> S_2 <n < t>$, siendo t una expresión entera.

Una estrategia conocida para la búsqueda del invariante p es la generalización de la postcondición q, reemplazando constantes de q por variables en p. Desde un punto de vista más abstracto, la construcción de un programa S totalmente correcto con respecto a la especificación <p, q> puede verse como el cálculo, dentro de H*, de la fórmula <p> S <q>. El cálculo se vale de la composicionalidad de H*. Y la sensatez y completitud de H* aseguran, respectivamente, la correctitud y la factibilidad de la construcción.

CAPÍTULO 3.3. VERIFICACIÓN DE PROGRAMAS SECUENCIALES NO DETERMINÍSTICOS

En este capítulo se extiende la metodología de verificación a los programas secuenciales no determinísticos. Dado que la programación concurrente también involucra no determinismo, este capítulo es además introductorio a la problemática de la verificación de programas concurrentes.

Ahora, a partir de un estado inicial puede haber más de una computación, y por lo tanto más de un estado final. En particular, puede haber computaciones que terminan y otras que no terminan. Se va a considerar fundamentalmente el caso en que el no determinismo es provocado por las estructuras de control: la selección condicional y la repetición. Otra posibilidad, que se incluye al final del capítulo, es la utilización de asignaciones aleatorias, capaces de asignar no determinísticamente un número natural a una variable.

La interpretación del no determinismo difiere del de la teoría de la computación, el que hemos considerado en las dos partes precedentes del libro: para que un programa satisfaga una propiedad, todas sus computaciones deben satisfacer la propiedad. Por ejemplo, todas las computaciones deben terminar. Las elecciones no determinísticas se resuelven de una manera arbitraria, no hay ninguna asunción de probabilidades (salvo si existe alguna hipótesis de fairness, que se define después). Esta interpretación prioriza la eficiencia de la implementación, dado que no hace necesario reproducir todas las posibles computaciones. La motivación por programar no determinísticamente pasa por otro lado, apunta esencialmente a la abstracción, es decir a evitar detalles de implementación que no tienen por qué especificarse en los programas.

La estructura de la presentación de los temas es como la del capítulo anterior. En la Sección 3.3.1 se describe la sintaxis y la semántica operacional del lenguaje de programación no determinístico con el que se va a trabajar. Se trata de GCL, o lenguaje de comandos con guardia (instrucciones precedidas por expresiones booleanas), que es una extensión de PLW con selecciones condicionales y repeticiones no determinísticas. En la semántica se agrega el estado indefinido de falla, que denota terminación incorrecta, la cual se produce cuando una selección condicional tiene todas sus guardias con valor falso. El estado de falla ya pudo haberse introducido en la semántica de PLW, considerando por ejemplo división por cero o índice fuera de rango.

En las Secciones 3.3.2 y 3.3.3 se describen los métodos de verificación de correctitud parcial y total, denominados D y D*, respectivamente. Se mantienen los principios generales discutidos en los métodos H y H* para la programación determinística. En concordancia con la semántica de GCL, ahora la acepción de terminación no sólo significa que todas las computaciones terminan, sino que además terminan sin estado de falla. Se presentan ejemplos de aplicación de los métodos.
En la Sección 3.3.4 se introduce el concepto de fairness (justicia o equidad), presente en distintos lenguajes de programación. El fairness establece restricciones sobre el comportamiento de las computaciones infinitas. Se definen dos tipos de fairness, débil y fuerte, que entonces impactan en la terminación de los programas de GCL. Consecuentemente, se modifica la regla del método D* relacionada con la terminación de la instrucción de repetición no determinística. Se presentan ejemplos de prueba con fairness débil y fuerte.
Finalmente, en la Sección 3.3.5 se adapta la definición de proof outline al nuevo lenguaje de programación, y se hace referencia al desarrollo sistemático de programas de GCL. Adicionalmente, se describe una regla de prueba para la asignación aleatoria, y se presenta una manera alternativa de probar programas de GCL con asunción de fairness. La idea es modificar el programa original agregándole asignaciones aleatorias, para luego desarrollar la prueba sin asunción de fairness.

3.3.1. Lenguaje de programación GCL

El lenguaje de programación que se utilizará para tratar la verificación de los programas no determinísticos se denomina GCL (por *Guarded Commands Language*, o lenguaje de comandos con guardia). GCL es una extensión no determinística del lenguaje PLW considerado en el capítulo anterior.

La sintaxis de un programa S ε GCL es la siguiente:

S :: skip | x := e | S_1 ; S_2 | [G] | *[G]
G :: B $\rightarrow$ S $\Box$ G | B $\rightarrow$ S

Las instrucciones skip, x := e, y S_1 ; S_2 , son las del lenguaje PLW. Las instrucciones [G] y *[G] son, respectivamente, la selección condicional no determinística y la repetición no determinística. La construcción B$\rightarrow$S

se conoce como comando con guardia booleana o directamente comando con guardia.
La selección condicional no determinística tiene, entonces, la forma:

$[B_1 \rightarrow S_1$
$\square$
$- - -$
$\square$
$B_n \rightarrow S_n]$.

Para simplificar la escritura, se la puede referenciar con la expresión $[\square_i B_i \rightarrow S_i]$, tal que i varía en un conjunto finito Δ de *direcciones*.

Informalmente, su semántica es: al momento de la ejecución de la instrucción de selección, se evalúan todas las guardias, de las que resultan verdaderas se elige no determinísticamente una guardia B_i, y luego se ejecuta la instrucción asociada S_i. Después, el programa continúa con la instrucción siguiente a la selección. Por otro lado, si ninguna de las guardias es verdadera, la selección, y el programa completo, terminan en un *estado de falla* (se dice que el programa *aborta* o *falla*).

En cuanto a la repetición no determinística, dicha instrucción tiene la siguiente forma:

$*[B_1 \rightarrow S_1$
$\square$
$- - -$
$\square$
$B_n \rightarrow S_n]$.

Para simplificar la escritura, se la puede referenciar con la expresión $*[\square_i B_i \rightarrow S_i]$, tal que $i \varepsilon \Delta$.

Informalmente, su semántica es: al momento de la ejecución de la instrucción de repetición, se evalúan todas las guardias, de las que resultan verdaderas se elige no determinísticamente una guardia B_i, y luego se ejecuta la instrucción asociada S_i. Después, se vuelven a evaluar las guardias, se elige una, se ejecuta la instrucción asociada, etc., y así siguiendo hasta que ninguna guardia sea verdadera, en cuyo caso el programa continúa con la instrucción siguiente a la repetición. Es decir que ahora, cuando ninguna de las guardias es verdadera, el

programa no aborta, sino que simplemente la repetición termina. Y en el caso de que siempre exista una guardia verdadera, la repetición, y el programa completo, *divergen*. Como en PLW, en el lenguaje GCL la repetición es la única instrucción que puede provocar divergencia en los programas.
Tanto en la selección como en la repetición, cuando una guardia B_i es verdadera se dice que la dirección i está *habilitada*. Notar que de acuerdo a la sintaxis de GCL, en ambas instrucciones se permite cualquier nivel de anidamiento de una y otra.

Ejemplo. El siguiente programa S_{num} de GCL devuelve en x algún número natural del intervalo [0..N]:

S_{num} :: x := 0 ; y := 0 ;
*[y < N $\rightarrow$ y := y + 1 ;
[true $\rightarrow$ x := y
$\square$
true $\rightarrow$ skip]]

Se presenta a continuación la semántica operacional del lenguaje GCL. Se omiten el skip, la asignación y la secuencia, porque ya se describieron en el capítulo anterior.
En este caso, la relación de transición «$\rightarrow$» describe transformaciones de configuraciones de la forma $C = \langle S_i, \sigma_i \rangle$, siendo S_i una continuación sintáctica de un programa S ε GCL:

(1) Selección condicional no determinística:
Si para algún i ε Δ vale $\sigma(B_i)$ = tt, entonces $\langle [\square_i B_i \rightarrow S_i], \sigma \rangle \rightarrow \langle S_i, \sigma \rangle$.
Si para todo i ε Δ vale $\sigma(B_i)$ = ff, entonces $\langle [\square_i B_i \rightarrow S_i], \sigma \rangle \rightarrow \langle E, f \rangle$.

Es decir, si hay más de una dirección i habilitada, se elige no determinísticamente alguna de ellas. La evaluación de las guardias no modifica el estado corriente y se consume en un paso.
Y si ninguna dirección i está habilitada, la instrucción termina en un estado de falla f, y el programa completo aborta. El estado f denota terminación incorrecta, y al igual que $\perp$ cumple: (a) Para toda aserción p, f $\not\models$ p. (b) Si $val(\pi(S_1, \sigma)) = f$, entonces $val(\pi(S_1 ; S_2, \sigma)) = f$ (la falla se propaga).

(2) Repetición no determinística:
Si para algún i ε Δ vale $\sigma(B_i)$ = tt, entonces $<*[\square_i B_i \rightarrow S_i], \sigma> \rightarrow <S_i ; *[\square_i B_i \rightarrow S_i], \sigma>$.
Si para todo i ε Δ vale $\sigma(B_i)$ = ff, entonces $<*[\square_i B_i \rightarrow S_i], \sigma> \rightarrow <E, \sigma>$.

Es decir, si hay más de una dirección i habilitada, se elige no determinísticamente alguna de ellas, para ejecutar la instrucción asociada S_i y después continuar con la ejecución de la repetición completa. La evaluación de las guardias no modifica el estado corriente y se consume en un paso. Y si ninguna dirección i está habilitada, la instrucción termina correctamente.

□

Por el no determinismo provocado por las instrucciones de selección condicional y repetición de GCL, ahora dado un programa S y un estado inicial σ, puede haber varias computaciones y varios estados finales, lo que se refleja en la figura 3.3.1:

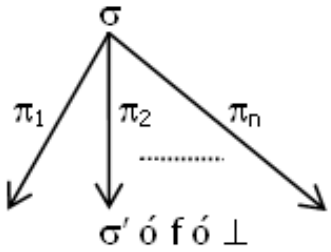


Figura 3.3.1

$\Pi(S, \sigma)$ denota el conjunto de todas las computaciones $\pi(S, \sigma)$, las cuales pueden *terminar*, *fallar* o *diverger*. En $\Pi(S, \sigma)$ pueden convivir computaciones de los tres tipos. El árbol de computaciones asociado a $\Pi(S, \sigma)$ se denota con $T_{S, \sigma}$. Los nodos de $T_{S, \sigma}$ representan las configuraciones de las distintas computaciones.

Ejemplo. El árbol $T_{num, \sigma}$ de la figura 3.3.2, con $\sigma \models N = 2$, es el árbol de computaciones asociado a $\Pi(S_{num}, \sigma)$, donde S_{num} es el programa de GCL presentado anteriormente. Para simplificar la figura, los nodos del árbol sólo muestran los estados.

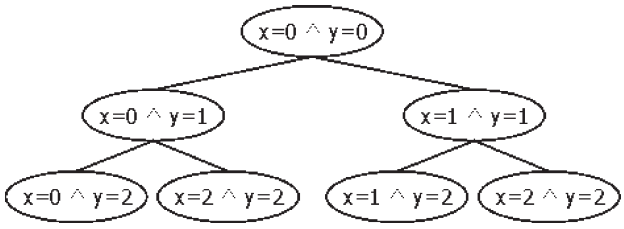


Figura 3.3.2

La función semántica asociada a GCL es:

$$\mathcal{M} : \text{GCL} \rightarrow (\Sigma \rightarrow \mathcal{P}(\Sigma)).$$

Ahora se utiliza el conjunto de partes $\mathcal{P}(\Sigma)$, porque el output de un programa $S \in$ GCL, a partir de un estado inicial $\sigma \in \Sigma$, es un conjunto de estados finales.
El conjunto de estados finales correspondiente a $\Pi(S, \sigma)$, representado en el árbol de computaciones $T_{S, \sigma}$ por sus hojas, se denota de la siguiente manera:

$$\mathcal{M}(S)(\sigma) = \{val(\pi(S, \sigma)) \mid \pi(S, \sigma) \in \Pi(S, \sigma)\}.$$

Puede suceder que $f \in \mathcal{M}(S)(\sigma)$, y también que $\perp \in \mathcal{M}(S)(\sigma)$.

Ejemplo. Dado el programa S_{num} y el estado inicial $\sigma \models N \geq 0$, se cumple que el conjunto de estados finales es: $\mathcal{M}(S_{num})(\sigma) = \{\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_N\}$, con $\sigma_i \models x = i$.
En este caso, todos los posibles estados finales son propios: $f \notin \mathcal{M}(S_{num})(\sigma)$, y $\perp \notin \mathcal{M}(S_{num})(\sigma)$.

Se cumple que para todo programa S y para todo estado σ, el conjunto de estados finales $\mathcal{M}(S)(\sigma)$ es finito o bien $\perp \varepsilon \mathcal{M}(S)(\sigma)$ (esta propiedad se conoce como *no determinismo acotado*). En otras palabras, la única posibilidad para que un programa S de GCL produzca un conjunto infinito de estados, es que una de sus computaciones sea infinita.
La propiedad de no determinismo acotado se prueba por el Lema de König, que establece que si un árbol de grado finito tiene un número infinito de nodos, entonces una de sus ramas es infinita. El árbol de computaciones $T_{S,\sigma}$ tiene grado finito porque los conjuntos Δ de direcciones son finitos, y entonces, si el número de hojas de $T_{S,\sigma}$ es infinito, significa que una de sus ramas $\pi(S, \sigma)$ es infinita.
Volviendo al programa S_{num}, notar que si se lo modifica para que produzca cualquier número natural, entonces alguna de sus computaciones debe diverger:

Ejemplo. El siguiente programa S_{num2} de GCL es una variante de S_{num}. A partir de un estado inicial σ |= x = 0, devuelve en la variable x algún número natural, ahora no acotado por una constante N. Se utiliza una variable booleana b:

S_{num2} :: *[b $\rightarrow$ x := x + 1
□
b $\rightarrow$ b := false]

Si S_{num2} termina, el valor de la variable x puede ser cualquier número natural. Pero S_{num2} puede diverger, porque puede darse el caso que siempre se elija la primera dirección de Δ.

De ahora en más será útil trabajar con variables booleanas, como en el ejemplo anterior. Se define:

Var = Ivar $\cup$ Bvar,

siendo Ivar el conjunto de variables enteras y Bvar el conjunto de variables booleanas. De esta manera, un estado σ deberá entenderse como un par de funciones:

σ_1: Ivar $\rightarrow$ V,

σ_2: Bvar $\rightarrow$ W = {tt, ff}.

□

A partir de la definición inductiva de la semántica de GCL presentada mediante la relación «$\rightarrow$», se pueden enumerar las distintas formas de las computaciones de los programas de GCL mediante el siguiente lema, como se hizo para el caso del lenguaje PLW:

Lema de las formas de las computaciones de los programas de GCL. Dada la configuración inicial $C_0 = <S_0, \sigma_0>$, las distintas formas de las computaciones $\pi(C_0)$ de $\Pi(C_0)$ son las siguientes (se omiten los casos del skip, la asignación y la secuencia, porque ya fueron descriptos antes, pero teniendo en cuenta que ahora en la secuencia puede haber falla además de divergencia):

1) Si S_0 :: [$\Box_i$ $B_i \rightarrow S_i$], las computaciones $\pi(C_0)$ tienen una de estas formas:

a) $C_0 \rightarrow <E, f>$, cuando al comienzo no vale ninguna guardia, por lo que la única computación $\pi(C_0)$ falla.

b) $C_0 \rightarrow C_1 = <S_i, \sigma_0> \rightarrow \ldots$, cuando al comienzo vale alguna guardia B_i, con $i \in \Delta$, y $<S_i, \sigma_0> \rightarrow \ldots$ es una computación del conjunto $\Pi(C_1)$ que termina, falla o diverge.

2) Si S_0 :: *[$\Box_i$ $B_i \rightarrow S_i$], las computaciones $\pi(C_0)$ tienen una de estas formas:

a) $C_0 \rightarrow <E, \sigma_0>$, cuando al comienzo no vale ninguna guardia, por lo que la única computación $\pi(C_0)$ termina.

b) $C_0 \rightarrow^* <S_0, \sigma_1> \rightarrow^* \ldots \rightarrow^* <S_0, \sigma_k> \rightarrow <S_i, \sigma_k> \rightarrow \ldots$, para algún $i \in \Delta$, cuando después de una cantidad finita de iteraciones se pasa a una computación $<S_i, \sigma_k> \rightarrow \ldots$ del conjunto $\Pi(<S_i, \sigma_k>)$ que falla o diverge.

c) $C_0 \rightarrow^* <S_0, \sigma_1> \rightarrow^* \ldots \rightarrow^* <S_0, \sigma_k> \rightarrow \ldots$, cuando en toda evaluación de las guardias siempre existe una que es verdadera, por lo que toda computación $\pi(C_0)$ de este tipo diverge.

d) $C_0 \rightarrow^* <S_0, \sigma_1> \rightarrow^* \ldots \rightarrow^* <E, \sigma_k>$, cuando después de una cantidad finita de iteraciones la evaluación de todas las guardias

resulta falsa, por lo que toda computación $\pi(C_0)$ de este tipo termina.

□

Dadas las consideraciones anteriores (varias computaciones, conjunto de estados finales, posibilidad de falla), en el marco de los programas GCL las propiedades de correctitud parcial y correctitud total ahora se definen de la siguiente manera:

Definición. Un programa S ε GCL es parcialmente correcto con respecto a la especificación <p, q>, es decir, $\models \{p\}\ S\ \{q\}$, sii:

$$[\forall \sigma, \sigma']: [(\sigma \models p \wedge \sigma' \varepsilon \mathcal{M}(S)(\sigma) \wedge \sigma' \neq f \wedge \sigma' \neq \perp) \rightarrow (\sigma' \models q)].$$

□

En palabras, un programa S de GCL es parcialmente correcto con respecto a <p, q>, sii a partir de cualquier estado σ que satisface la precondición p, toda computación de S que termina lo hace en un estado σ' que satisface la postcondición q.
La correctitud parcial no se viola si existe un estado σ que satisface p a partir del cual alguna computación de S falla o diverge. Tampoco se viola cuando se consideran estados σ que no satisfacen p, independientemente de lo que suceda al final de las computaciones correspondientes.

Definición. Un programa S ε GCL es totalmente correcto con respecto a la especificación <p, q>, es decir, $\models \langle p \rangle\ S\ \langle q \rangle$, sii:

$$[\forall \sigma, \sigma']: [(\sigma \models p \wedge \sigma' \varepsilon \mathcal{M}(S)(\sigma)) \rightarrow (\sigma' \neq f \wedge \sigma' \neq \perp \wedge \sigma' \models q)].$$

□

En palabras, un programa S de GCL es totalmente correcto con respecto a <p, q>, sii a partir de cualquier estado σ que satisface p, toda computación de S termina en un estado σ' que satisface q.
La correctitud total no se viola cuando se consideran estados σ que no satisfacen p, independientemente de lo que suceda al final de las computaciones correspondientes.

Sigue valiendo el Lema de Separación, es decir:

$\models \langle p \rangle\ S\ \langle q \rangle \leftrightarrow (\models \{p\}\ S\ \{q\} \ \wedge\ \models \langle p \rangle\ S\ \langle true \rangle)$.

Ejercicio. *Probar el Lema de Separación en el marco de los programas S de GCL.*

□

Además de las instrucciones de selección condicional y repetición, otra instrucción de GCL que puede introducir no determinismo es la *asignación aleatoria*, que tiene la forma x := ?, y cuya semántica informal es que la variable x recibe cualquier número natural.
El no determinismo provocado por la selección condicional y la repetición se conoce como no determinismo *guiado por el control* (*control driven*), mientras que el que introducen las asignaciones aleatorias se denomina no determinismo *guiado por los datos* (*data driven*). En las siguientes secciones se tratará solamente el no determinismo guiado por el control. Al final del capítulo se volverá a las asignaciones aleatorias, cuando se trate el *fairness*.

3.3.2. Método D para verificar la correctitud parcial de los programas de GCL

En esta sección se presenta el método de verificación D para probar, en el marco de GCL, las fórmulas de correctitud parcial {p} S {q}, es decir que un programa S de GCL sea parcialmente correcto con respecto a <p, q>.
Los axiomas ASI y SKIP, y las reglas SEC y CONS, son los del método H. Las reglas siguientes D-COND y D-REP corresponden, respectivamente, a las instrucciones de selección condicional y repetición. Como se verá, mantienen las mismas ideas que las reglas homólogas del método H:

(1) Regla de la selección condicional no determinística (D-COND): dadas las aserciones p y q, las expresiones booleanas B_i y los subprogramas S_i , con i variando en el conjunto Δ de direcciones,

$$\frac{\{p \wedge B_i\}\ S_i\ \{q\}}{\{p\}\ [\Box_i\ B_i \rightarrow S_i]\ \{q\}}$$

La regla establece que si para todos los casos de las direcciones i $\varepsilon\ \Delta$, se cumple la postcondición q cuando la ejecución de S_i termina a partir

de $(p \wedge B_i)$, entonces se cumple q cuando la ejecución de $[\Box_i\ B_i \rightarrow S_i]$ termina a partir de la precondición p.

(2) Regla de la repetición no determinística (D-REP): dada la aserción p, las expresiones booleanas B_i y los subprogramas S_i , con i variando en el conjunto Δ de direcciones,

$$\frac{\{p \wedge B_i\}\ S_i\ \{p\}}{\{p\}\ *[\Box_i\ B_i \rightarrow S_i]\ \{p \wedge (\bigwedge_i \neg B_i)\}}$$

La regla establece que si para todos los casos de las direcciones $i \in \Delta$, la ejecución de S_i cuando termina preserva p a partir de la condición $(p \wedge B_i)$, entonces se cumple que a partir de p, la ejecución de $*[\Box_i\ B_i \rightarrow S_i]$ cuando termina preserva p, valiendo además $(\bigwedge_i \neg B_i)$. La aserción p es el invariante de la repetición.

□

En el ejempo siguiente se presenta una prueba utilizando el método D:

Ejemplo. Con el siguiente programa S_{div} de GCL se pretende obtener en la variable y algún número natural divisor de $x \geq 1$:

S_{div} :: z := 1 ; y := 1 ;
*[z < x → z := z + 1 ; [z | x → [true → y := z □ true → skip]
□
¬ z | x → skip]]

Se va a probar $\vdash_D \{x \geq 1\}\ S_{div}\ \{y \mid x\}$. La expresión y | x denota que y divide a x. Se propone como invariante de la repetición de S_{div} la aserción p = (y | x). Para simplificar la escritura de la prueba, a la repetición incluida en S_{div} se la denotará con S_{rep}, y a la selección condicional [z | x → [true → y := z □ true → skip] □ ¬ z | x → skip] con S_{sel} .

El plan de la prueba es el siguiente:

(a) $\vdash \{x \geq 1\}\ z := 1 ; y := 1\ \{y \mid x\}$,
(b) $\vdash \{y \mid x\}\ S_{rep}\ \{y \mid x\}$,

para llegar finalmente a:

(c) $\vdash \{x \geq 1\}\ S_{div}\ \{y \mid x\}$, por la aplicación de la regla SEC sobre (a) y (b).

Prueba de (a):

(1) $\{x \geq 1\}\ z := 1 ; y := 1\ \{y \mid x\}$	(ASI, SEC, CONS)

Prueba de (b):

(2) $\{(y \mid x \wedge z \mid x) \wedge true\}\ y := z\ \{y \mid x\}$	(ASI, CONS)
(3) $\{(y \mid x \wedge z \mid x) \wedge true\}\ skip\ \{y \mid x\}$	(SKIP, CONS)
(4) $\{y \mid x \wedge z \mid x\}\ [true \rightarrow y := z \square true \rightarrow skip]\ \{y \mid x\}$	(2, 3, D-COND)
(5) $\{y \mid x \wedge \neg z \mid x\}\ skip\ \{y \mid x\}$	(SKIP, CONS)
(6) $\{y \mid x\}\ [z \mid x \rightarrow [true \rightarrow y := z \square true \rightarrow skip] \square \neg z \mid x \rightarrow skip]\ \{y \mid x\}$	(4, 5, D-COND)
(7) $\{y \mid x \wedge z < x\}\ z := z + 1\ \{y \mid x\}$	(ASI, CONS)
(8) $\{y \mid x \wedge z < x\}\ z := z + 1 ; S_{sel}\ \{y \mid x\}$	(6, 7, SEC)
(9) $\{y \mid x\}\ S_{rep}\ \{y \mid x \wedge \neg (z < x)\}$	(8, D-REP)
(10) $\{y \mid x\}\ S_{rep}\ \{y \mid x\}$	(9, CONS)

Se cumple que el método de verificación D es sensato, es decir que para todo par de aserciones p y q, y para todo programa S de GCL, vale:

$$Tr \vdash_D \{p\}\ S\ \{q\} \quad \rightarrow \quad \models \{p\}\ S\ \{q\}.$$

Como en el caso del método H, para la prueba de sensatez de D se utiliza inducción sobre la longitud de la prueba, demostrando que los axiomas son verdaderos y que las reglas son sensatas.
Las pruebas de sensatez de las reglas D-COND y D-REP son muy similares a las de las reglas homólogas de H, y quedan como ejercicio para el lector.
También se cumple que D es relativamente completo, es decir que para todo par de aserciones p y q, y para todo programa S de GCL, vale:

$$\models \{p\}\ S\ \{q\} \quad \rightarrow \quad Tr \vdash_D \{p\}\ S\ \{q\}.$$

Se asume que el lenguaje de especificación Assn es expresivo. Como en el caso de H, para demostrar la completitud de D se prueba, utilizando inducción estructural, que cualquiera sea la forma del programa S de

GCL, si se cumple $\models \{p\}\ S\ \{q\}$ entonces se puede probar $\{p\}\ S\ \{q\}$ en D. La prueba para el caso de la selección condicional es muy similar a la que se hizo para H, y queda como ejercicio para el lector.
También la prueba para el caso de la repetición es muy similar a la que se hizo para H. Para probar que si $\models \{r\}\ *[\Box_i B_i \rightarrow S_i]\ \{q\}$, entonces $Tr \vdash_D \{r\}\ *[\Box_i B_i \rightarrow S_i]\ \{q\}$, hay que encontrar un invariante p que cumpla:

(a) $r \rightarrow p$, (b) para todo $i \in \Delta$: $\models \{p \wedge B_i\}\ S_i\ \{p\}$, y (c) $(p \wedge \bigwedge_i \neg B_i) \rightarrow q$,

porque aplicando la hipótesis inductiva se llega a:

para todo $i \in \Delta$: $\vdash \{p \wedge B_i\}\ S_i\ \{p\}$,

y finalmente por la aplicación de las reglas D-REP y CONS se obtiene:

$$\vdash \{r\}\ *[\Box_i B_i \rightarrow S_i]\ \{q\}.$$

Ahora, el invariante p debe denotar el conjunto de todos los estados que se obtienen a partir de la precondición r, a lo largo de las n iteraciones (pueden ser infinitas) de S_i, con $i \in \Delta$, tal que los S_i pueden variar de iteración en iteración según la elección no determinística que se haga de la dirección habilitada. Es decir,

$$p = p_0 \vee \ldots \vee p_k \vee \ldots,$$

siendo $p_0 = r$, y $p_{m+1} = \bigvee_i post(p_m \wedge B_i, S_i)$ con $i \in \Delta$.

Las aserciones $post(p_m \wedge B_i, S_i)$ existen por la expresividad de Assn, y por el mismo motivo se puede probar que la disyunción infinita $\bigvee_i p_i$ es expresable en Assn mediante una aserción finita p.

3.3.3. Método D* para verificar la correctitud total de los programas de GCL

En esta sección se presenta el método de verificación D* para probar, en el marco de GCL, las fórmulas de correctitud total <p> S <q>, es decir que un programa S de GCL sea totalmente correcto con respecto a la especificación <p, q>.

Los axiomas y reglas del método D* son los de D, reemplazando los símbolos «{ }» por «< >» y agregando al final del nombre del axioma o regla el símbolo «*», a excepción de las dos reglas relacionadas con las instrucciones no determinísticas (selección condicional y repetición) porque son las únicas que pueden causar no terminación (respectivamente, falla o divergencia). Se presentan a continuación estas dos reglas:

(1) Regla D-COND*: dadas las aserciones p y q, las expresiones booleanas B_i y los subprogramas S_i , con i variando en el conjunto Δ de direcciones,

$$\frac{p \rightarrow \bigvee_i B_i \ , \ <p \wedge B_i> \ S_i \ <q>}{<p> \ [\ \square_i \ B_i \rightarrow S_i \ ] \ <q>}$$

Es decir, se le agrega a la regla D-COND de correctitud parcial una premisa para asegurar que la selección no falle: si vale la precondición p, debe existir alguna guardia B_i que sea verdadera.

(2) Regla D-REP*: dada la aserción p, las expresiones booleanas B_i y los subprogramas S_i , con i variando en el conjunto Δ de direcciones,

$$\frac{\begin{array}{l}\text{INIC:} \quad p \rightarrow \exists n: p(n) \\ \text{CONT:} \quad p(n+1) \rightarrow \bigvee_i B_i \\ \text{DEC:} \quad <p(n) \wedge n > 0 \wedge B_i> \ S_i \ <\exists k: k < n \wedge p(k)> \\ \text{TERM:} \quad p(0) \rightarrow \bigwedge_i \neg B_i\end{array}}{<p> \ *[\ \square_i \ B_i \rightarrow S_i \ ] \ <p(0)>}$$

Como se ve, la regla D-REP* mantiene las mismas ideas que la regla homóloga del método H*.
Se sigue utilizando un invariante p, que la ejecución de todo S_i preserva cuando se ejecuta a partir de la condición ($p \wedge B_i$), y por eso p sigue valiendo al final de la repetición completa. Y se le agrega un parámetro natural n que no forma parte de las variables del programa S y que representa la máxima cantidad posible de iteraciones.
La condición inicial (INIC) establece que la precondición p, es decir el invariante, asegura la existencia del parámetro natural n. Notar que en

la regla REP* de H* la expresión $\exists$n: p(n) aparece como precondición de la conclusión (es una formulación equivalente).
La condición de continuación (CONT) establece que mientras $n \neq 0$, alguna guardia B_i es verdadera. Recién cuando n = 0, todas las guardias B_i son falsas, que es lo que establece la condición de terminación (TERM).
Finalmente, la regla de decremento (DEC) establece que la ejecución de todo S_i reduce el valor del parámetro n en una o más unidades, es decir que cualquiera sea la dirección i elegida en cada iteración, n se decrementa.
Por lo tanto, se mantiene la idea de asociar a toda repetición el conjunto parcialmente ordenado bien fundado (N, <). Al no existir cadenas descendentes infinitas, se llega a la postcondición p(0), y como p(0) $\rightarrow \bigwedge_i \neg B_i$, entonces por la semántica de GCL la repetición termina.
Se puede formular una regla D-REP* alternativa más flexible, que no exija que al final el valor de n sea cero, para permitir pruebas más sencillas.

□

Ejercicio. *Probar que el programa anterior S_{div}, que ya se demostró es parcialmente correcto con respecto a* $<x \geq 1, y \mid x>$, *no aborta, considerando la misma especificación (se puede utilizar la prueba desarrollada previamente).*

□

En el ejemplo siguiente se desarrolla una prueba utilizando la regla D-REP*:

Ejemplo. El siguiente programa S_{num3} de GCL es otra variante de S_{num} para devolver, dado el estado inicial $\sigma \models x = 0 \wedge N \geq 0$, algún número natural del intervalo [0..N].

S_{num3} :: *[b $\wedge$ x < N $\rightarrow$ x := x + 1
□
b $\rightarrow$ b := false].

Se va a probar en D*:

<x = 0 $\wedge$ N $\geq$ 0> S_{num3} <true>.

Se propone como invariante parametrizado de la repetición, la aserción p(n) siguiente:

$p(n) = [(b \wedge x < N \rightarrow n = N - x + 1) \wedge (b \wedge x \geq N \rightarrow n = 1) \wedge (\neg b \rightarrow n = 0)]$.

Esta forma de aserción, con casos asociados a las distintas guardias de la repetición, es típica en estas pruebas. El variante decreciente n representa la «distancia» a la terminación. A continuación se prueban, una a una, las premisas de D-REP*. Tener en cuenta en lo que sigue que $n \varepsilon N$:

Premisa INIC. Debe cumplirse:

$[x = 0 \wedge N \geq 0] \rightarrow [\exists n: (b \wedge x < N \rightarrow n = N - x + 1) \wedge (b \wedge x \geq N \rightarrow n = 1) \wedge (\neg b \rightarrow n = 0)]$.

Si $\neg b$, entonces $n = 0$. Si $b \wedge x \geq N$, entonces $n = 1$. Y si $b \wedge x < N$, entonces $n = N - x + 1$, con $n \varepsilon N$ porque $n \geq 2$. Por lo tanto, vale la implicación.

Premisa CONT. Debe cumplirse:

$[(b \wedge x < N \rightarrow n + 1 = N - x + 1) \wedge (b \wedge x \geq N \rightarrow n + 1 = 1) \wedge (\neg b \rightarrow n + 1 = 0)] \rightarrow [(b \wedge x < N) \vee b]$.

Si se cumple el antecedente, entonces no vale $\neg b$ porque $n \varepsilon N$. En cambio puede valer $b \wedge x < N$ (sería $n \geq 1$), o bien $b \wedge x \geq N$ (sería $n = 0$), es decir que vale b. Por lo tanto, vale la implicación.

Premisa TERM. Debe cumplirse:

$[(b \wedge x < N \rightarrow 0 = N - x + 1) \wedge (b \wedge x \geq N \rightarrow 0 = 1) \wedge (\neg b \rightarrow 0 = 0)] \rightarrow [\neg (b \wedge x < N) \wedge \neg b]$.

Si se cumple el antecedente, entonces no vale $b \wedge x < N$ ni $b \wedge x \geq N$. La única posibilidad es que valga $\neg b$. Por lo tanto, vale la implicación.

Premisa DEC. Debe cumplirse:

(a) $<p(n) \wedge n > 0 \wedge b \wedge x < N>$ $x := x + 1$ $<\exists k: k < n \wedge p(k)>$.
(b) $<p(n) \wedge n > 0 \wedge b>$ $b := false$ $<\exists k: k < n \wedge p(k)>$.

Caso (a):

(a1) Por ASI*:
$<\exists k: k < n \wedge (b \wedge x + 1 < N \rightarrow k = N - (x + 1) + 1) \wedge (b \wedge x + 1 \geq N \rightarrow k = 1) \wedge (\neg b \rightarrow k = 0)>$
$x := x + 1$
$<\exists k: k < n \wedge (b \wedge x < N \rightarrow k = N - x + 1) \wedge (b \wedge x \geq N \rightarrow k = 1) \wedge (\neg b \rightarrow k = 0)>$

(a2) Por MAT:
$[(b \wedge x < N \rightarrow n = N - x + 1) \wedge (b \wedge x \geq N \rightarrow n = 1) \wedge (\neg b \rightarrow n = 0) \wedge n > 0 \wedge b \wedge x < N] \rightarrow$
$[\exists k: k < n \wedge (b \wedge x + 1 < N \rightarrow k = N - (x + 1) + 1) \wedge (b \wedge x + 1 \geq N \rightarrow k = 1) \wedge (\neg b \rightarrow k = 0)]$.
Si se cumple el antecedente, entonces $n > 0 \wedge b \wedge x < N \wedge n = N - x + 1$. Hay dos casos. Si $x + 1 < N$, entonces $k = n - 1$, y por lo tanto $k < n$ y $k \varepsilon N$, porque $n \geq 3$. Y si $x + 1 = N$, entonces $k = 1$ y $n = 2$, y por lo tanto otra vez $k < n$ y $k \varepsilon N$.

Aplicando CONS* sobre (a1) y (a2) se llega a lo que se pretendía.

Caso (b):

(b1) Por ASI*:
$<\exists k: k < n \wedge (\text{false} \wedge x < N \rightarrow k = N - x + 1) \wedge (\text{false} \wedge x \geq N \rightarrow k = 1) \wedge (\neg \text{false} \rightarrow k = 0)>$
$b := \text{false}$
$<\exists k: k < n \wedge (b \wedge x < N \rightarrow k = N - x + 1) \wedge (b \wedge x \geq N \rightarrow k = 1) \wedge (\neg b \rightarrow k = 0)>$

(b2) Por MAT:
$[(b \wedge x < N \rightarrow n = N - x + 1) \wedge (b \wedge x \geq N \rightarrow n = 1) \wedge (\neg b \rightarrow n = 0) \wedge n > 0 \wedge b] \rightarrow$
$[\exists k: k < n \wedge (\text{false} \wedge x < N \rightarrow k = N - x + 1) \wedge (\text{false} \wedge x \geq N \rightarrow k = 1) \wedge (\neg \text{false} \rightarrow k = 0]$.
El consecuente se simplifica a $\exists k: k < n \wedge k = 0$ y se cumple, porque si se cumple el antecedente, vale $n > 0$.

Aplicando CONS* sobre (b1) y (b2) otra vez se llega a lo que se pretendía.

Como se cumplen las cuatro premisas, vale $<x = 0 \wedge N \geq 0> S_{num3} <p(0)>$, y como $p(0) \rightarrow \text{true}$, entonces finalmente: $<x = 0 \wedge N \geq 0> S_{num3} <\text{true}>$.

Ejercicio. *Completar la prueba de* $<x = 0 \wedge N \geq 0> S_{num3} <0 \leq x \leq N>$*, es decir, recurriendo al Lema de Separación probar* $\{x = 0 \wedge N \geq 0\} S_{num3} \{0 \leq x \leq N\}$*.*

□

Se cumple que el método de verificación D* es sensato. Para todo par de aserciones p y q, y para todo programa S de GCL, vale:

$$Tr \vdash_{D^*} <p> S <q> \rightarrow \models <p> S <q>.$$

Las pruebas de sensatez de las reglas D-COND* y D- REP* son muy similares a las de las reglas homólogas del método H*, aún teniendo en cuenta:

- el agregado de la premisa $p \rightarrow \vee_i B_i$ en la regla D-COND*, y que
- en D-REP* el decremento del variante n no es uniforme, puede ser de una o más unidades, y se produce cualquiera sea la dirección habilitada que se elija.

Ejercicio. *Probar la sensatez de las reglas D-COND* y D-REP*.*

□

También se cumple que D* es relativamente completo. Para todo par de aserciones p y q, y para todo programa S de GCL, vale:

$$\models <p> S <q> \rightarrow Tr \vdash_{D^*} <p> S <q>.$$

3.3.4. Verificación de no divergencia en GCL asumiendo *fairness*

Hasta ahora no se ha asumido ningún tipo de *fairness* (equidad o justicia) en la semántica del lenguaje GCL, es decir, una dirección $i \in \Delta$ habilitada frecuentemente en una repetición no determinística $*[\Box_i B_i \rightarrow S_i]$ no tiene por qué ser elegida alguna vez.

Por ejemplo, en el programa S_1 de GCL:

S_1 :: x := 0 ; b := true;
*[1: b $\rightarrow$ x := x + 1
□
2: b $\rightarrow$ b := false],

una posible computación π de S_1 a partir de cualquier estado σ, es la que siempre pasa por la dirección 1, lo que provoca divergencia (las direcciones 1 y 2 insertas en S_1 no son parte de la sintaxis de GCL, sólo se utillizan para hacer más claras las explicaciones). Si bien la dirección 2 está permanentemente habilitada, sin fairness puede suceder que nunca sea elegida.
Si la semántica de GCL asegurara que una dirección de una repetición, habilitada permanentemente a lo largo de una computación infinita, no puede ser postergada indefinidamente, el programa anterior terminaría, porque alguna vez deberá elegirse entonces la dirección 2. Este tipo de fairness se denomina *fairness débil* (enseguida se verá por qué se denomina de este modo). Obviamente, con fairness débil las direcciones habilitadas permanentemente en una computación infinita son elegidas infinitas veces.

Notar que en el programa siguiente S_2 de GCL, aún con fairness débil puede haber divergencia:

S_2 :: x := 0 ; b := true;
*[1: b $\rightarrow$ x := x + 1
□
2: b $\wedge$ par(x) $\rightarrow$ b := false].

La función par(x) es verdadera cuando el valor de x es par. Otra vez, una posible computación divergente es la que a partir de cualquier estado siempre pasa por la dirección 1. No se viola el fairness débil porque la dirección 2 no está permanentemente habilitada, sino sólo de manera intermitente.
Si la semántica de GCL asegurara que una dirección de una repetición, habilitada infinitas veces a lo largo de una computación infinita, no puede ser postergada indefinidamente, el programa anterior terminaría, porque alguna vez se elegirá entonces la dirección 2. Este tipo de fairness se denomina *fairness fuerte*. Con fairness fuerte, las direcciones habilitadas infinitas veces en una computación infinita son elegidas infinitas veces. Las calificaciones de débil y fuerte del fairness, se deben a que si la semántica de GCL contempla el fairness fuerte, entonces por definición también contempla el fairness débil.

Ejercicio. *Probar que si hay fairness fuerte, entonces hay fairness débil. ¿Se cumple la recíproca?*

□

El siguiente ejemplo muestra un programa S_3 de GCL, en el que puede haber divergencia aún con fairness fuerte:

S_3 :: x := 1 ;
*[1: x > 0 → x := x + 1
□
2: x > 0 → x := x – 1].

Por ejemplo, existe la computación infinita $\pi(S_3, \sigma)$, con $\sigma \models$ true, que pasa de manera alternada por las direcciones 1 y 2, lo que no viola el fairness fuerte.
En términos del árbol de computaciones $T_{S, \sigma}$, el fairness implica la «poda» de computaciones que no son *fair*, es decir, computaciones infinitas que violan el tipo de fairness contemplado.
Para denotar que un programa S es totalmente correcto con respecto a <p, q> asumiendo fairness débil o fuerte, se utilizan, respectivamente, las expresiones:

$\models$ <<p>>^{w} S <<q>>^{w} (w por *weak* o débil), o bien

$\models$ <<p>>^{s} S <<q>>^{s} (s por *strong* o fuerte).

También en estos casos sigue valiendo el Lema de Separación (la prueba queda como ejercicio para el lector):

a) $\models$ <<p>>^{w} S <<q>>$^w \leftrightarrow$ ($\models$ {p} S {q} $\wedge$ $\models$ <<p>>^{w} S <<true>>^{w}).
b) $\models$ <<p>>^{s} S <<q>>$^s \leftrightarrow$ ($\models$ {p} S {q} $\wedge$ $\models$ <<p>>^{s} S <<true>>^{s}).

El ejemplo presentado a continuación muestra que ahora, con fairness, deja de valer el no determinismo acotado. Sea el programa siguiente, y supóngase que hay fairness débil:

S_1 :: x := 0 ; b := true;
*[1: b → x := x + 1
□
2: b → b := false].

El programa S_1 termina siempre a partir de cualquier estado inicial σ, y produce como output cualquier número natural. Es decir, $\mathcal{M}(S_1)(\sigma) = \{\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n, \ldots\}$, con $\sigma \models$ true y $\sigma_i \models x = i$, y en este caso se cumple que $\perp \notin \mathcal{M}(S_1)(\sigma)$ (para indicar que hay fairness, la función semántica se puede denotar con $\mathcal{M}^w$ ó $\mathcal{M}^s$, según sea el caso).

Para la verificación de no divergencia de los programas de GCL, ahora con fairness no se va a requerir que el variante se decremente cualquiera sea la dirección elegida, sino que alcanzará con que esto suceda sólo con determinadas direcciones útiles. La correctitud parcial y la ausencia de falla no se impactan porque con el fairness solamente se descartan computaciones infinitas (por definición, todas las computaciones finitas son fair). En términos más generales, con fairness se ven afectadas solamente las propiedades de tipo *liveness*, como la no divergencia, y la ausencia de inanición en el caso de los programas concurrentes.
Se presenta primeramente la regla WD-REP*, que reemplaza a D-REP*, para la verificación de no divergencia en los programas de GCL asumiendo fairness débil. Esta aproximación se denomina *método de las direcciones útiles.* Con el objeto de simplificar las siguientes formulaciones y ejemplos, de ahora en más no se permitirán repeticiones no determinísticas anidadas.

Se pretende probar $<p>>^w *[\Box_i B_i \rightarrow S_i] <<true>>^w$. Dada la repetición $S :: *[\Box_i B_i \rightarrow S_i]$, si:

a) (W, <) es un conjunto parcialmente ordenado bien fundado, con minimal w_0,
b) la aserción p(w) es un invariante parametrizado,
c) para todo $w \varepsilon W$, con $w > w_0$, $\Delta_w = \Delta^d_w \cup \Delta^s_w$ es una partición del conjunto Δ de direcciones, con $\Delta^d_w \neq \emptyset$, y
d) se cumplen las siguientes premisas:

INIC: $p \rightarrow \exists w: p(w)$
CONT: $(\forall w > w_0)(\exists i \varepsilon \Delta^d_w) : p(w) \rightarrow B_i$
DEC: $<<p(w) \wedge w > w_0 \wedge B_i>>^w S_i <<\exists v \varepsilon W: v < w \wedge p(v)>>^w$, para todo $i \varepsilon \Delta^d_w$
NOINC: $<<p(w) \wedge w > w_0 \wedge B_i>>^w S_i <<\exists v \varepsilon W: v \leq w \wedge p(v)>>^w$, para todo $i \varepsilon \Delta^s_w$
TERM: $p(w_0) \rightarrow \bigwedge_i \neg B_i$, con $i \varepsilon \Delta$,

entonces: $<<p>>^w *[\Box_i B_i \rightarrow S_i] <<p(w_0)>>^w$.

□

Se cumple que el método WD*, es decir D* reemplazando D-REP* por WD-REP*, es sensato y relativamente completo.

Informalmente, la sensatez de la regla WD-REP* se debe a que:

- Por cada valor del variante distinto del minimal w_0, existe un conjunto no vacío de direcciones útiles, que acortan la distancia a la terminación porque decrementan el valor del variante. Y existe otro conjunto de direcciones, que puede ser vacío, que no perjudican (mantienen o decrementan el valor del variante). Los dos conjuntos de direcciones son, respectivamente, Δ^d_w y Δ^s_w. La premisa DEC establece que todas las direcciones de Δ^d_w decrementan el variante, y la premisa NOINC, que ninguna dirección de Δ^s_w lo incrementa.
- La premisa CONT asegura que mientras no se alcanza el minimal, existe una determinada guardia B_i que es verdadera (por lo que la repetición no termina), siendo i una dirección útil. Al haber fairness débil, la dirección i será elegida alguna vez, y entonces el variante se va a decrementar necesariamente.
- Las premisas INIC y TERM son las mismas premisas de la regla D-REP*, salvo que ahora hacen referencia en general a un conjunto parcialmente ordenado bien fundado, que puede ser distinto de (N, <). Por INIC, el invariante p asegura que existe un valor inicial para el parámetro w, que representa la distancia a la terminación, y por TERM, cuando se alcanza el minimal la repetición termina.

Formalmente, supóngase que $Tr \vdash_{WD^*} \langle\langle p \rangle\rangle^w\ ^*[\Box_i B_i \to S_i]\ \langle\langle true \rangle\rangle^w$, pero que existe una computación infinita $\pi = \langle^*[\Box_i B_i \to S_i], \sigma_0\rangle \to^* \langle^*[\Box_i B_i \to S_i], \sigma_1\rangle \to^* \langle^*[\Box_i B_i \to S_i], \sigma_2\rangle \to^* \ldots$

Dado el estado inicial $\sigma \models p$, por INIT se cumple que $\exists w: p(w)$.

Por TERM, y dado que la repetición no termina, debe ser $w > w_0$.

Por CONT, DEC, NOINC, y dado que la repetición no termina, existe una secuencia infinita de elementos de W asociada a π, cuya parte final (infinita) tiene la forma $w_1 = w_2 = \ldots = w_n = \ldots$, siendo todo w_k distinto del minimal w_0. Esto significa que a partir de un momento dado, no se aplica más la regla DEC, lo que viola el fairness débil dado que por CONT, vale $\exists i\ \varepsilon\ \Delta^d_w: B_i$.

□

En el siguiente ejemplo se presenta una prueba que utiliza la regla WD-REP*:

Ejemplo. Probar $<<b_1 \wedge b_2 \wedge x = 0 \wedge y = 0>>^w$ S $<<true>>^w$, con:

S :: *[1: $b_1 \wedge b_2 \rightarrow x := x + 1$
□
2: $b_1 \wedge b_2 \rightarrow b_1 := false$
□
3: $\neg b_1 \wedge b_2 \rightarrow y := y + 1$
□
4: $\neg b_1 \wedge b_2 \rightarrow b_2 := false$]

Se propone:

- El orden (W = {0, 1, 2}, <) con la relación «<» habitual.
- $p(w) = ((b_1 \wedge b_2 \rightarrow w = 2) \wedge (\neg b_1 \wedge b_2 \rightarrow w = 1) \wedge (\neg b_1 \wedge \neg b_2 \rightarrow w = 0) \wedge (b_1 \rightarrow b_2))$.
- Y las particiones $\Delta^d_2 = \{2\}$ y $\Delta^s_2 = \{1, 3, 4\}$ para w = 2, y $\Delta^d_1 = \{4\}$ y $\Delta^s_1 = \{1, 2, 3\}$ para w = 1.

Informalmente:

- El valor inicial w = 2 representa la distancia a la terminación.
- Al comienzo valen permanentemente las guardias B_1 y B_2. Por el fairness débil, en algún momento se elegirá la dirección útil 2, por lo que el variante pasará a valer 1 y las guardias B_3 y B_4 serán verdaderas permanentemente.
- Finalmente, otra vez por el fairness débil, en algún momento se elegirá la dirección útil 4, por lo que el variante obtendrá el valor minimal 0 y el programa terminará.
- De esta manera el variante descenderá dos «escalones», asociados a los valores 2 y 1, hasta alcanzar definitivamente el valor cero.

Formalmente:

La aserción p(2) se simplifica en $(b_1 \wedge b_2)$.
La aserción p(1) se simplifica en $(\neg b_1 \wedge b_2)$.
La aserción p(0) se simplifica en $(\neg b_1 \wedge \neg b_2)$.

INIC. Se cumple: $(b_1 \wedge b_2 \wedge x = 0 \wedge y = 0) \rightarrow p(2)$.

CONT. Se cumple: $p(2) \rightarrow B_2$, con $2 \in \Delta^d_2$, y se cumple $p(1) \rightarrow B_4$, con $4 \in \Delta^d_1$.

TERM. Se cumple: $p(0) \rightarrow (\neg (b_1 \wedge b_2) \wedge \neg (\neg b_1 \wedge b_2))$.

DEC.
a) Se cumple: $\langle\langle p(2) \wedge b_1 \wedge b_2 \rangle\rangle^w b_1 := \text{false} \langle\langle p(1) \rangle\rangle^w$, porque $(p(2) \wedge b_1 \wedge b_2) \rightarrow (\neg \text{false} \wedge b_2)$.
b) Se cumple $\langle\langle p(1) \wedge \neg b_1 \wedge b_2 \rangle\rangle^w b_2 := \text{false} \langle\langle p(0) \rangle\rangle^w$, porque $(p(1) \wedge \neg b_1 \wedge b_2) \rightarrow (\neg b_1 \wedge \neg \text{false})$.

NOINC.
a) Se cumple: $\langle\langle p(2) \wedge b_1 \wedge b_2 \rangle\rangle^w x := x + 1 \langle\langle p(2) \rangle\rangle^w$, porque $(p(2) \wedge b_1 \wedge b_2) \rightarrow p(2)$.
b) Se cumple: $\langle\langle p(2) \wedge \neg b_1 \wedge b_2 \rangle\rangle^w y := y + 1 \langle\langle p(2) \rangle\rangle^w$, porque $(p(2) \wedge \neg b_1 \wedge b_2) \rightarrow p(2)$.
c) Se cumple: $\langle\langle p(2) \wedge \neg b_1 \wedge b_2 \rangle\rangle^w b_2 := \text{false} \langle\langle p(2) \rangle\rangle^w$, porque $(p(2) \wedge \neg b_1 \wedge b_2) \rightarrow (b_1 \wedge \text{false})$.
d) Se cumple: $\langle\langle p(1) \wedge b_1 \wedge b_2 \rangle\rangle^w x := x + 1 \langle\langle p(1) \rangle\rangle^w$, porque $(p(1) \wedge b_1 \wedge b_2) \rightarrow p(1)$.
e) Se cumple: $\langle\langle p(1) \wedge b_1 \wedge b_2 \rangle\rangle^w b_1 := \text{false} \langle\langle p(1) \rangle\rangle^w$, porque $(p(1) \wedge b_1 \wedge b_2) \rightarrow (\neg \text{false} \wedge b_2)$.
f) Se cumple: $\langle\langle p(1) \wedge \neg b_1 \wedge b_2 \rangle\rangle^w y := y + 1 \langle\langle p(1) \rangle\rangle^w$, porque $(p(1) \wedge \neg b_1 \wedge b_2) \rightarrow p(1)$.

Por lo tanto, por WD-REP* vale $\langle\langle b_1 \wedge b_2 \wedge x = 0 \wedge y = 0 \rangle\rangle^w S \langle\langle p(0) \rangle\rangle^w$, y así:

$$\langle\langle b_1 \wedge b_2 \wedge x = 0 \wedge y = 0 \rangle\rangle^w S \langle\langle \text{true} \rangle\rangle^w.$$

Para la verificación de no divergencia en los programas de GCL asumiendo fairness fuerte se puede emplear la regla SD-REP*, que también se basa en la idea de direcciones útiles. Se cumple que el método SD* (D* reemplazando D-REP* por SD-REP*) es sensato y relativamente completo.

Se pretende probar $\langle p \rangle\rangle^s *[\Box_i B_i \rightarrow S_i] \langle\langle \text{true} \rangle\rangle^s$. Dada la repetición $S :: *[\Box_i B_i \rightarrow S_i]$, si:

a) (W, <) es un conjunto parcialmente ordenado bien fundado, con minimal w_0,
b) la aserción p(w) es un invariante parametrizado,
c) para todo $w \varepsilon W$, con $w > w_0$, $\Delta_w = \Delta^d_w \cup \Delta^s_w$ es una partición del conjunto Δ de direcciones, con $\Delta^d_w \neq \varnothing$, y
d) se cumplen las siguientes premisas:

INIC: $p \rightarrow \exists w: p(w)$

CONT: $p(w) \wedge w > w_0 \rightarrow \vee_i B_i$, con $i \in \Delta$

DEC: $<<p(w) \wedge w > w_0 \wedge B_i>>^s S_i <<\exists v \in W: v < w \wedge p(v)>>^s$, para todo $i \in \Delta^d_w$

NOINC: $<<p(w) \wedge w > w_0 \wedge B_i>>^s S_i <<\exists v \in W: v \leq w \wedge p(v)>>^s$, para todo $i \in \Delta^s_w$

HAB-INF: $<<p(w) \wedge w > w_0>>^s S'_w :: *[\Box_k (B_k \wedge_m \neg B_m) \rightarrow S_k] <<true>>^s$, con $k \in \Delta^s_w$ y $m \in \Delta^d_w$

TERM: $p(w_0) \rightarrow \wedge_i \neg B_i$, con $i \in \Delta$,

entonces: $<<p>>^s *[\Box_i B_i \rightarrow S_i] <<p(w_0)>>^s$.

□

La regla SD-REP* es similar a WD-REP*. La diferencia radica en que tiene una nueva premisa y en que CONT sólo establece que mientras el variante no alcanza el minimal w_0, entonces alguna dirección, sea útil o no perjudicial, está habilitada (la repetición sigue). La nueva premisa HAB-INF (por habilitaciones infinitas) asegura que por el fairness fuerte, el variante se irá decrementando necesariamente:

- HAB-INF se irá aplicando recursivamente, cada vez sobre un programa derivado S'_w con menos direcciones, porque las k direcciones no perjudiciales no son todas debido a que el conjunto Δ^d_w no es vacío. Cuando k = 0, se asume que S'_w = skip.
- El programa derivado S'_w termina con fairness fuerte por uno de dos motivos. O bien ninguna guardia B_k es verdadera, y por lo tanto eventualmente sólo quedan direcciones útiles m habilitadas. O bien alguna guardia B_m es verdadera, lo que significa que existe una dirección útil m habilitada, y por lo tanto no puede existir una computación infinita que pase solamente por direcciones no perjudiciales k dado que se estaría violando el fairness fuerte.

Ejemplo. Volviendo al programa:

S_2 :: x := 0 ; b := true;
*[1: b → x := x + 1
□
2: b ∧ par(x) → b := false],

podría definirse, para probar <<true>>^{s} S_2 <<true>>^{s} mediante la regla SD-REP*:

- El orden (W = {0, 1}, <) con la relación «<» habitual,
- p(w) = (b $\rightarrow$ w = 1) $\wedge$ ($\neg$ b $\rightarrow$ w = 0),
- y la partición Δ^d_1 = {2} y Δ^s_1 = {1} para w = 1,

quedando el programa derivado:

$$S'_1 :: *[b \wedge \neg (b \wedge par(x)) \rightarrow x := x + 1],$$

el cual es determinístico y termina a lo sumo después de una iteración.

Ejercicio. *Completar la prueba del ejemplo.*

□

Otra variante del método de las direcciones útiles para probar no divergencia en GCL con fairness, que no se va a presentar, se basa en la utilización de aserciones en lugar de «distancias» a la terminación, para caracterizar las direcciones útiles.
En la siguiente sección se hace referencia a otra aproximación, basada en las asignaciones aleatorias.

3.3.5. Otras consideraciones

3.3.5.1. Proof outlines en el marco de los programas de GCL

Las proof outlines estándar de correctitud total se definen de la misma manera que la que se vio para H*. Ahora, para el caso de los subprogramas T :: [$\Box_i$ $B_i \rightarrow S_i$], deben cumplirse las siguientes implicaciones entre las aserciones intercaladas:

$pre(T) \rightarrow \vee_i B_i$, $pre(T) \wedge B_i \rightarrow pre(S_i)$, y $post(S_i) \rightarrow post(T)$, para todo $i \varepsilon \Delta$.

Y para el caso de los subprogramas T :: *[$\Box_i$ $B_i \rightarrow S_i$], debe cumplirse:

$pre(T) \wedge B_i \rightarrow pre(S_i)$, $pre(T) \wedge \wedge_i \neg B_i \rightarrow post(T)$, y $post(S_i) \rightarrow pre(T)$, para todo $i \varepsilon \Delta$.

□

Por ejemplo, se presenta a continuación una proof outline de correctitud total de:

$<x = X \wedge y = Y \wedge X > 0 \wedge Y > 0> S_{mcd} <x = y \wedge y = mcd(X, Y)>$,

siendo S_{mcd} un programa de GCL que calcula el máximo común divisor de dos números naturales mayores que cero. La expresión mcd(X, Y) denota el máximo común divisor de los números naturales X e Y. Se define como invariante, $p = mcd(x, y) = (mcd(X, Y) \wedge x > 0 \wedge y > 0)$, y como variante (en el dominio N), n = x + y:

$<x = X \wedge y = Y \wedge X > 0 \wedge Y > 0>$
$<inv: mcd(x, y) = mcd(X, Y) \wedge x > 0 \wedge y > 0, fc: x + y>$
S_{mcd} :: $*[x > y \rightarrow <mcd(x, y) = mcd(X, Y) \wedge x > 0 \wedge y > 0 \wedge x > y>$
x := x – y
$<mcd(x, y) = mcd(X, Y) \wedge x > 0 \wedge y > 0>$
□
$x < y \rightarrow <mcd(x, y) = mcd(X, Y) \wedge x > 0 \wedge y > 0 \wedge x < y>$
y := y – x
$<mcd(x, y) = mcd(X, Y) \wedge x > 0 \wedge y > 0>]$
$<mcd(x, y) = mcd(X, Y) \wedge x > 0 \wedge y > 0 \wedge \neg (x > y) \wedge \neg (x < y)>$
$<x = y \wedge y = mcd(X, Y)>$

Notar que en este caso, la proof outline no es estándar, y el variante no alcanza el minimal cero.

3.3.5.2. Desarrollo sistemático de programas de GCL

Las características del desarrollo sistemático de programas mencionadas en el capítulo anterior, también aplican para GCL.
La construcción de un programa correcto de GCL con respecto a una especificación <p, q>, puede verse como el cálculo en D* (o en WD*, o en SD*, según la hipótesis de fairness) de la fórmula <p> S <q>. Se cuenta con la propiedad de composicionalidad. Y por la sensatez y completitud relativa del método, se puede asegurar la correctitud y la factibilidad de la construcción, respectivamente.
Para desarrollar sistemáticamente un programa S de GCL de la forma típica:

$S :: S_0 ; *[\Box_i B_i \rightarrow S_i]$, con $i \varepsilon \Delta$,

que satisfaga la especificación <r, q>, deben encontrarse y posteriormente implementarse en términos de las instrucciones de GCL, un invariante p y un variante decreciente w. Asumiendo que se utiliza (N, <), que no es necesario que el variante alcance el valor minimal cero, y que no hay fairness, debe cumplirse:

a) $<r> S_0 <p>$.
b) $<p \wedge B_i> S_i <p>$, para todo $i \varepsilon \Delta$.
c) $p \wedge \bigwedge_i \neg B_i \rightarrow q$, con $i \varepsilon \Delta$.
d) $p \rightarrow w \geq 0$.
e) $<p \wedge B_i \wedge w = t> S_i <w < t>$, para todo $i \varepsilon \Delta$, siendo t una expresión entera.

La estrategia típica para la obtención del invariante p es la generalización de la postcondición q, reemplazando constantes de q por variables en p.

3.3.5.3. Asignaciones aleatorias

Una instrucción adicional del lenguaje GCL a considerar es la *asignación aleatoria*, ya referida previamente. Las asignaciones aleatorias introducen el no determinismo a través de los datos.
Sea el lenguaje ampliado $GCL_{AA} = GCL \cup \{x := ?\}$, tal que x := ? es la sintaxis de la asignación aleatoria. La semántica informal de dicha instrucción establece que la variable x recibe cualquier número natural. Formalmente, la semántica se define mediante la relación «$\rightarrow$» de la siguiente manera:

$<x := ?, \sigma> \rightarrow <E \mid \sigma[x \mid n]>$, con $n \varepsilon N$.

□

Como sucede con el fairness, con el uso de las asignaciones aleatorias el no determinismo no es acotado. Notar que el árbol de computaciones $T_{x := ?, \sigma}$ no tiene grado finito. Esta similitud entre el fairness y las asignaciones aleatorias se puede explotar para la verificación de no divergencia asumiendo fairness, como se verá enseguida.

Para la verificación de programas de GCL_{AA} se debe definir adicionalmente, entonces, el axioma de la asignación aleatoria (ASI_{AA}):

Para toda aserción p, $\{\forall x: p\}\ x := ?\ \{p\}$.

□

Además, para la prueba de no divergencia ahora no se puede utilizar el dominio de los números naturales, porque la cantidad de iteraciones de una repetición no depende del estado inicial, es decir, no está acotada. Por ejemplo, sea el siguiente programa S de GCL_{AA}, ejecutado a partir de cualquier estado σ:

$$
\begin{aligned}
S :: *[\ & b \wedge x > 0 \rightarrow x := x - 1 \\
& \square \\
& b \wedge x < 0 \rightarrow x := x + 1 \\
& \square \\
& \neg\, b \rightarrow x := ?\ ;\ b := true\ ].
\end{aligned}
$$

Si b es verdadero inicialmente, S termina al cabo de |x| iteraciones. En cambio, si b es falso, no se puede predecir el número de iteraciones del programa, porque este valor se conoce recién cuando se haya efectuado la asignación aleatoria. Por lo tanto, el variante asociado a la repetición debería expresarse como:

$n = t \geq |x|$, para todo $x \geq 0$,

siendo t una expresión entera, lo cual es imposible. Se debe recurrir a otro conjunto parcialmente ordenado bien fundado. La solución consiste en trabajar con el dominio $W = N \cup \{\omega\}$, tal que ω es mayor que todo número natural, es decir, es el primer ordinal infinito.
Una posible regla para probar no divergencia en GCL_{AA} (regla D-REP$^*_{AA}$) es la que se presenta a continuación. Dado el dominio W recién definido, la aserción p, las expresiones booleanas B_i y los subprogramas S_i, con i variando en el conjunto Δ de direcciones,

$$
\begin{aligned}
&\text{INIC:} && p \rightarrow \exists \alpha: p(\alpha) \\
&\text{CONT:} && p(\alpha) \wedge \alpha > 0 \rightarrow \bigvee_i B_i \\
&\text{DEC:} && \langle p(\alpha) \wedge \alpha > 0 \wedge B_i \rangle\ S_i\ \langle \exists \beta: \beta < \alpha \wedge p(\beta) \rangle \\
&\text{TERM:} && p(0) \rightarrow \bigwedge_i \neg\, B_i \\
&\text{———————————————} \\
& && \langle p \rangle\ *[\ \square_i\ B_i \rightarrow S_i\ ]\ \langle p(0) \rangle
\end{aligned}
$$

□

Un interesante uso de las asignaciones aleatorias es para probar no divergencia asumiendo fairness de una manera alternativa al método de las direcciones útiles. La idea es, dado un programa S de GCL:

- Primero transformar S en un programa S' de GCL_{AA}, ampliándolo con asignaciones aleatorias, con el objetivo de introducir explícitamente un *scheduler* que sea fair, es decir, que anule las computaciones que no sean fair.
- Luego verificar que S' no diverge recurriendo al método D^*_{AA} (D* ampliado con ASI^*_{AA} y reemplazando D-REP* por $D\text{-}REP^*_{AA}$).

Esta aproximación alternativa al método de las direcciones útiles se conoce como *método del scheduler explícito*, y se ejemplifica a continuación.
Sea un esquema de programa de GCL muy simple, con sólo dos direcciones:

$S :: *[1: B_1 \rightarrow S_1 \; \square \; 2: B_2 \rightarrow S_2]$.

Introduciéndole asignaciones aleatorias para implementar una política de fairness fuerte, el programa transformado S' de GCL_{AA} quedaría de la siguiente manera:

$$
\begin{aligned}
S' :: \; & z_1 := ? \; ; \; z_2 := ? \; ; \\
& *[1: B_1 \wedge z_1 \leq z_2 \rightarrow S_1 \; ; \; z_1 := ? \; ; \; [B_2 \rightarrow z_2 := z_2 - 1 \; \square \neg B_2 \rightarrow skip] \\
& \quad \square \\
& \quad 2: B_2 \wedge z_2 < z_1 \rightarrow S_2 \; ; \; z_2 := ? \; ; \; [B_1 \rightarrow z_1 := z_1 - 1 \; \square \neg B_1 \rightarrow skip]].
\end{aligned}
$$

Es decir, se introducen dos variables que no están en S, las variables z_1 y z_2, por medio de las cuales se implementa la política del scheduler fair. Las variables z_1 y z_2 representan las prioridades asignadas a las direcciones 1 y 2, respectivamente, de manera tal que el decremento de z_i implica el aumento de la prioridad de la dirección i.
Como se observa, después de atravesar la dirección i, la prioridad de la otra dirección (dirección k) aumenta, siempre que esté habilitada a pesar de no haber sido elegida. Al mismo tiempo, la prioridad de i se reinicializa. El decremento gradual de z_k cuando k está habilitada, asegura que esta dirección no será postergada indefinidamente.
Al comienzo, las variables z_1 y z_2 se inicializan con valores arbitrarios no negativos. Dada z_i, con i = 1 ó 2, el valor $z_i + 1$ representa el número máximo de iteraciones que pueden transcurrir antes que la dirección i sea elegida, contando sólo las veces en que i está habilitada. Lo mismo sucede después de las reinicializaciones aleatorias de las variables.

CAPÍTULO 3.4. VERIFICACIÓN DE PROGRAMAS CONCURRENTES

El último capítulo sobre verificación de programas trata la concurrencia. Los programas concurrentes consisten en un conjunto de procesos secuenciales que se ejecutan concurrentemente, comunicándose y sincronizándose entre sí. El control de los programas se sitúa en distintos lugares al mismo tiempo, uno por cada proceso.

Se considera tanto el modelo de comunicación por variables compartidas, como el modelo de comunicación por pasajes de mensajes (en este caso, los programas se denominarán programas *distribuidos*, porque sus procesos no comparten variables). Por la diversidad de temas, cada uno es tratado suscintamente. El objetivo central sigue siendo mostrar cómo las ideas generales de verificación se instancian sobre cada tipo de programa, destacando las novedades, para presentar la metodología de una manera incremental. Al mismo tiempo, se sigue remarcando la idea subyacente de considerar los axiomas y reglas de los métodos como guías para el desarrollo sistemático de programas, que en el caso de la concurrencia, por su singular complejidad, adquieren un interés práctico especial.

Como en el caso de la programación no determinística, a partir de un estado inicial puede haber más de una computación, y así más de un estado final. Esto se debe a la semántica de intercalación (*interleaving*) no determinística de las instrucciones atómicas de los distintos procesos, característica de la concurrencia. Para un mismo programa pueden existir computaciones finitas y computaciones infinitas. En particular, las computaciones finitas pueden terminar en el estado de falla si es que se produce deadlock, es decir, si uno o más procesos están bloqueados indefinidamente, por haber quedado a la espera de un evento que nunca se producirá (a partir de un mal uso de las primitivas de sincronización).

La correctitud total incluye, además de la correctitud parcial y la no divergencia, la propiedad de ausencia de deadlock. Otras propiedades a considerar serían la ausencia de inanición (o ausencia de starvation), es decir que todo proceso que compita por un recurso, a futuro lo obtenga; y la exclusión mutua, es decir que ningún proceso manipule inadecuadamente variables que comparta con otro. La correctitud parcial, la ausencia de deadlock y la exclusión mutua, pertenecen a la familia de propiedades safety, y se prueban inductivamente. La no divergencia y la ausencia de inanición, pertenecen a la familia de propiedades liveness, y se prueban en base a variantes decrecientes de conjuntos parcialmente ordenados bien fundados.

Se verá que en la concurrencia deja de cumplirse la propiedad de composicionalidad. Esto determina que las pruebas de los programas concurrentes se basen en las proof outlines introducidas previamente. Las pruebas se desarrollan en dos etapas. En la primera etapa se prueban los procesos secuenciales, elaborándose una proof outline para cada uno. En la segunda etapa se consisten las proof outlines, con un criterio de consistencia que depende del modelo de concurrencia y el lenguaje de programación utilizados. Se desarrolla una prueba distinta por propiedad.

Otra característica de la verificación de los programas concurrentes es la necesidad de utilizar variables auxiliares, sin las cuales los métodos no serían completos. El objetivo de dichas variables es reforzar, en el sentido lógico, las aserciones de las proof outlines. Son variables auxiliares porque no afectan el cómputo básico.

En la Sección 3.4.1 se trata la verificación de los programas concurrentes con variables compartidas, y en la Sección 3.4.2, la verificación de los programas distribuidos.

En la Subsección 3.4.1.1 se introduce un primer lenguaje concurrente con variables compartidas, el lenguaje SVL. SVL tiene una primitiva de sincronización muy elemental, que determina que las pruebas de los programas sean muy desestructuradas.

En la Subsección 3.4.1.2 se describe el método de verificación de correctitud parcial de los programas de SVL, denominado O, y se presentan ejemplos de aplicación. En el método O, el criterio de consistencia entre las proof outlines secuenciales es que éstas sean *libres de interferencia*.

En la Subsección 3.4.1.3 se describe el método de verificación de correctitud total de los programas de SVL, denominado O*, y se presentan ejemplos de aplicación. Se trata la prueba de ausencia de deadlock y la de ausencia de divergencia. Cada prueba requiere primero obtener proof outlines secuenciales de los procesos. No se considera ninguna hipótesis de fairness, para facilitar la presentación.

En la Subsección 3.4.1.4 se introduce un segundo lenguaje concurrente con variables compartidas, el lenguaje RVL. RVL provee exclusión mutua en el nivel de las variables. Las variables compartidas se distribuyen entre recursos, sobre los que los procesos tienen acceso exclusivo. Esto facilita las pruebas de los programas. Los recursos constituyen una versión primitiva de los monitores.

En la Subsección 3.4.1.5 se describen los métodos de verificación de correctitud parcial y total de los programas de RVL, denominados respectivamente R y R*, y se presentan ejemplos de aplicación. Una

característica de los mismos es la utilización de invariantes asociados a los recursos, para expresar la relación entre las variables de los recursos cuando éstos están libres. La consistencia entre las proof outlines se establece en base a los invariantes de los recursos.
En la Subsección 3.4.2.1 se introduce el lenguaje distribuido con que se va a trabajar, (un fragmento de) CSP, que es una extensión concurrente del lenguaje no determinístico GCL. El pasaje de mensajes es sincrónico: se sincroniza el envío de un mensaje de un proceso con la recepción del mensaje de otro proceso.
En la Subsección 3.4.2.2 se describe el método de verificación de correctitud parcial de los programas de CSP, denominado AFR, y se presentan ejemplos de aplicación. En el método AFR, el criterio de consistencia entre las proof outlines secuenciales se denomina *cooperación*. Otra característica es que se deben utilizar invariantes globales, junto con variables auxiliares, para asegurar la completitud del método.
Por último, en la Subsección 3.4.2.3 se describe el método de verificación de correctitud total de los programas de CSP. Se trata la prueba de ausencia de deadlock y la de ausencia de divergencia, con ejemplos. Cada prueba requiere primero obtener proof outlines secuenciales de los procesos. Como antes, no se considera ninguna hipótesis de fairness, para facilitar la presentación. Al final de la subsección se hace mención al modelo asincrónico de pasajes de mensajes.

3.4.1. Verificación de programas concurrentes con variables compartidas

3.4.1.1. Lenguaje SVL

El primer lenguaje con el que se va a trabajar es SVL (por *Shared Variables Language* o lenguaje de variables compartidas).
Un programa P ε SVL es un programa concurrente con variables compartidas que tiene la forma:

$$P :: P_0 ; [P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n].$$

El símbolo «||» denota la composición concurrente. El subprograma P_0 pertenece a PLW. Cada proceso P_i, con $1 \leq i \leq n$, es un proceso secuencial compuesto por un subprograma S_i con instrucciones de PLW

o bien una nueva instrucción, la instrucción await, que se va a describir enseguida. P_0 contiene inicializaciones a variables. Para simplificar la presentación, no se permite el anidamiento de la composición concurrente.
Los programas se podrán denotar de la siguiente manera:

$$P :: P_0 ; [||_{i = 1,n} P_i], \text{ o bien directamente } P :: P_0 ; [|| P_i].$$

En algunos ejemplos, para facilitar las descripciones, se utilizarán etiquetas (*labels*) para identificar a los procesos.
Informalmente, la ejecución de un programa P de SVL consiste en la ejecución de un subprograma P_0, seguida de la ejecución concurrente de procesos secuenciales P_i. Hay varias computaciones posibles, según cómo se intercalen en su ejecución, no determinísticamente, las instrucciones atómicas de los procesos (por eso a esta semántica se la conoce como *modelo de intercalación* o *interleaving*). Por lo tanto, como en el caso de la programación no determinística, se trata con un conjunto de estados finales.
Por ejemplo, si <S> denota la ejecución atómica del subprograma S, el esquema de programa de SVL siguiente:

$$P :: [P_1 :: <S_1> ; <S_2> \;||\; P_2 :: <S_3> ; <S_4> ; <S_5> ; <S_6>]$$

podría ejecutarse como la secuencia:

$\pi_1 = <S_1> ; <S_2> ; <S_3> ; <S_4> ; <S_5> ; <S_6>$, o bien la secuencia:
$\pi_2 = <S_3> ; <S_4> ; <S_5> ; <S_6> ; <S_1> ; <S_2>$, o bien la secuencia:
$\pi_3 = <S_1> ; <S_3> ; <S_2> ; <S_4> ; <S_5> ; <S_6>$, etc.

Los procesos se comunican a través de variables compartidas. De esta manera, el contenido de una variable compartida en un momento dado depende de las asignaciones que se le hayan ejecutado desde distintos procesos. Esta característica provoca, como se verá cuando se trate la verificación de la correctitud parcial de los programas de SVL, que deje de cumplirse en los procesos secuenciales la preservación composicional referida en los capítulos precedentes: ahora la postcondición post(T) de un subprograma T de S, no sólo depende de la ejecución de T, sino también de la ejecución de subprogramas de otros procesos que modifiquen variables que utiliza S.
La instrucción await es la primitiva de sincronización de SVL. Un proceso ejecuta un await para producir un retardo (*delay*) que asegure el

cumplimiento de un determinado criterio de consistencia asociado a una o más variables compartidas, o bien para obtener exclusividad sobre una sección crítica (sección de programa con variables compartidas modificables).

La sintaxis del await es:

await B $\rightarrow$ S end,

siendo B una guardia booleana y S un subprograma de PLW (para simplificar la presentación, no se permite el anidamiento de instrucciones await).
Informalmente, la ejecución del await consiste en evaluar la guardia booleana B, y si resulta verdadera, se puede ejecutar el subprograma S completo, es decir que la instrucción await es atómica. Si en cambio B resulta falsa, el proceso P_i en el que se encuentra el await queda *bloqueado*, hasta que tras una evaluación posterior de la guardia B, si resulta verdadera, el await se puede consumir y en consecuencia el proceso puede progresar.
Formalmente, la semántica operacional del await se define de la siguiente manera:

Si $\sigma \models B$ y $\langle S, \sigma\rangle \rightarrow^* \langle E, \sigma'\rangle$, entonces $\langle$await B $\rightarrow$ S end, $\sigma\rangle \rightarrow \langle E, \sigma'\rangle$.

□

La utilización del await quedará más clara cuando se complete la definición de la relación «$\rightarrow$».

Ahora, la relación «$\rightarrow$» debe considerar configuraciones tanto secuenciales como concurrentes. Estas últimas tienen la forma $\langle[\ ||\ S_i], \sigma\rangle$, siendo las S_i, con $1 \leq i \leq n$, continuaciones sintácticas de los procesos P_i, y σ un estado global único con variables de todos los procesos. La relación «$\rightarrow$» entre configuraciones concurrentes se define de la siguiente manera:

Si $\langle S_i, \sigma\rangle \rightarrow \langle S'_i, \sigma'\rangle$, entonces $\langle[S_1 \,||\, \ldots \,||\, S_i \,||\, \ldots \,||\, S_n], \sigma\rangle \rightarrow_{\langle i, \sigma\rangle} \langle[S_1 \,||\, \ldots \,||\, S'_i \,||\, \ldots \,||\, S_n], \sigma'\rangle$.

El indicador $\langle i, \sigma\rangle$ denota que se avanza un paso por P_i, con estado corriente σ. En general, se define:

$$<[\,||\, S_i\,], \sigma> \rightarrow^*_h <[\,||\, S'_i\,], \sigma'>,$$

donde $h = <i_1,\sigma><i_2,\sigma_2>\ldots<i_k,\sigma_k>$ es la *historia* de las transformaciones de configuraciones concurrentes desde $<[\,||\, S_i\,], \sigma>$ hasta $<[\,||\, S'_i\,], \sigma'>$: primero se avanza por $P_{i\,1}$ dado σ, luego por $P_{i\,2}$ dado σ_2, y así hasta llegar, al cabo de k pasos, a la configuración $<[\,||\, S'_i\,], \sigma'>$.

□

La expresión $\Pi(P, \sigma)$ denota el conjunto de todas las computaciones posibles $\pi(P, \sigma)$, y el conjunto de estados finales correspondiente se denota con $\mathcal{M}(P)(\sigma) = \{val(\pi(P, \sigma)) \mid \pi(P, \sigma) \in \Pi(P, \sigma)\}$, siendo:

$$\mathcal{M} : SVL \rightarrow (\Sigma \rightarrow \mathcal{P}(\Sigma)),$$

la función semántica asociada a SVL.

Una computación concurrente $\pi(P, \sigma)$:

- *Termina*, si $\pi = <[\,||\, S_i\,], \sigma> \rightarrow^* <[\,||\, E_i\,], \sigma'>$.
- *Falla*, si π es finita y no termina. En este caso se dice que el programa tiene o está en *deadlock* (bloqueo mutuo o abrazo mortal), lo que se denota con $f \in \mathcal{M}(P, \sigma)$, siendo f el estado de falla.
- *Diverge*, si π es infinita, lo que se denota con $\perp \in \mathcal{M}(P, \sigma)$.

Un programa P de SVL es parcialmente correcto con respecto a <p, q>, es decir $\models \{p\}\ P\ \{q\}$, sii a partir de cualquier $\sigma \models p$, toda computación $\pi(P, \sigma)$, si termina, lo hace en un estado $\sigma' \models q$.

Y un programa P de SVL es totalmente correcto con respecto a <p, q>, es decir $\models <p>\ P\ <q>$, sii a partir de cualquier $\sigma \models p$, toda computación $\pi(P, \sigma)$ termina en un estado $\sigma' \models q$.

Se presenta a continuación un primer ejemplo de programa de SVL:

Ejemplo. El siguiente programa P_{pc} de SVL, implementa una solución concurrente al clásico problema del productor-consumidor:

P_{pc} :: P_0 :: in := 0 ; out := 0 ; i := 1 ; k := 1 ;
[productor :: while i ≤ n do
x := A[i] ; await in – out < b → skip end ; buffer[in mod b] := x ;
in := in + 1 ; i := i + 1
od
||
consumidor :: while k ≤ n do
await in – out > 0 → skip end ; y := buffer[out mod b] ;
out := out + 1 ; B[k] := y ; k := k + 1
od]

El programa copia los n elementos de un arreglo A a un arreglo B, por intermedio de un *buffer* de tamaño b. Las instrucciones await se utilizan para sincronización: (a) El productor chequea que haya espacio en el buffer, y si no lo hay se queda esperando. (b) El consumidor chequea que el buffer tenga elementos, y si no los tiene se queda esperando. Las operaciones sobre los índices del arreglo buffer son módulo b.

3.4.1.2. Verificación de la correctitud parcial de los programas de SVL

Para probar la correctitud parcial de un programa P de SVL con respecto a una especificación <p, q>, se va a utilizar el método de verificación O. El método O incluye los axiomas y reglas de H, más dos nuevas reglas, una para la instrucción await, y la otra para la composición concurrente (en realidad, se verá después que para que O sea relativamente completo, hará falta incluir una regla más).
La regla para el await (AWAIT) se define de la siguiente manera:

$$\frac{\{p \wedge B\}\ S\ \{q\}}{\{p\}\ \text{await } B \rightarrow S \text{ end } \{q\}}$$

Se prueba fácilmente que la regla AWAIT es sensata (queda como ejercicio para el lector). No se hace referencia a la posibilidad del retardo condicional de la instrucción (que provoca un bloqueo en el proceso correspondiente), porque la ausencia de deadlock se trata en la verificación de la correctitud total.

□

En lo que hace a la composición concurrente, una regla natural para la misma sería:

$$\frac{\{p_i\}\ S_i\ \{q_i\}\ ,\ 1 \le i \le n}{\{\wedge_{i=1,n}\ p_i\}\ [\ ||_{i=1,n}\ S_i\ ]\ \{\wedge_{i=1,n}\ q_i\}}$$

Pero esta regla no es sensata, como lo muestra el siguiente ejemplo. Se cumple:
(a) $\models \{x = 0\}\ P_1 :: \text{await true} \rightarrow x := x + 1 \text{ end}; y := 0; \text{await true} \rightarrow x := x - 1 \text{ end } \{x = 0 \wedge y = 0\}$,
(b) $\models \{x = 0\}\ Q :: z := x\ \{x = 0 \wedge z = 0\}$,

y así, dada la precondición x = 0, la postcondición de $[P_1 \,||\, Q]$ es:

$x = 0 \wedge y = 0 \wedge (z = 0 \vee z = 1)$, distinta de $x = 0 \wedge y = 0 \wedge z = 0$.

Notar además que dado:

(c) $\models \{x = 0\}\ P_2 :: y := 0\ \{x = 0 \wedge y = 0\}$,

en cambio ahora sí se cumple, con la precondición x = 0, que la postcondición de $[P_2 \,||\, Q]$ es:

$x = 0 \wedge y = 0 \wedge z = 0$.

Es decir, P_1 y P_2 no son intercambiables a pesar de ser equivalentes desde el punto de vista funcional, compuestos concurrentemente con Q pueden producir outputs distintos. En otras palabras, en la concurrencia deja de valer la propiedad de composicionalidad.
La pérdida de la composicionalidad hace que los programas concurrentes dejen de tratarse como «cajas negras». Ahora deberán considerarse los estados intermedios y las aserciones que los denotan.
Para definir una regla sensata asociada a la composición concurrente, se recurre a las proof outlines estándar de correctitud parcial introducidas previamente. Por lo tanto, primeramente habrá que completar la definición de proof outline estándar para el caso de la instrucción await:

Dado S :: await $B \rightarrow T$ end, entonces:

$pre(S) \wedge B \rightarrow pre(T)$, y $post(T) \rightarrow post(S)$.

Para simplificar la escritura, de ahora en más se va a omitir la calificación de estándar de las proof outlines, entendiendo que todas las proof outlines que se consideren lo serán.
Además de las *proof outlines secuenciales*, que son las que se definieron hasta ahora, se necesitará introducir el concepto de *proof outline concurrente*:

Definición. Una proof outline concurrente es un conjunto de proof outlines secuenciales *libres de interferencia*, en el siguiente sentido. Dado $P :: [\ ||\ P_i\ ]$, con $P_i :: S_i$, $1 \leq i \leq n$, si:

- para todo par de proof outlines secuenciales de $\{p_i\}\ S_i\ \{q_i\}$ y $\{p_k\}\ S_k\ \{q_k\}$, denotadas con S^*_i y S^*_k, respectivamente,
- para toda aserción r de S^*_i no interna de un await (se denomina *aserción normal*),
- y para toda instrucción S de S^*_k, siendo S un await o una asignación no interna de un await (se denomina *asignación normal*),

se cumple $\{r \wedge pre(S)\}\ S\ \{r\}$, y además $p \rightarrow \bigwedge_{i = 1,n} p_i$ y $\bigwedge_{i = 1,n} q_i \rightarrow q$, entonces el conjunto de proof outlines secuenciales de $\{p_i\}\ S_i\ \{q_i\}$ es una proof outline concurrente de $\{p\}\ P\ \{q\}$.

□

Es decir, con las proof outlines concurrentes la idea es forzar la preservación composicional de las proof outlines secuenciales (por eso se las referencia como proof outlines secuenciales libres de interferencia). Esto se logra asegurando que las aserciones de una proof outline secuencial no puedan invalidarse por la ejecución de ninguna instrucción de otro proceso que modifique variables compartidas, que sólo pueden ser el await o las asignaciones normales. De esta manera, ahora se puede introducir una regla sensata para la composición concurrente (la regla O-CONC):

$$\frac{\{p_i\}\ S^*_i\ \{q_i\},\ 1 \leq i \leq n,\ \text{son proof outlines secuenciales libres de interferencia}}{\{\bigwedge_{i = 1,n} p_i\}\ [\ ||_{\ i = 1,n}\ S_i\ ]\ \{\bigwedge_{i = 1,n} q_i\}}$$

□

Notar que la premisa de la regla O-CONC no se define en términos de fórmulas de correctitud, sino de proof outlines. En este sentido, O-CONC es una meta-regla.
También notar que a pesar de que no se cumple la composicionalidad, se mantiene la idea de componer objetos probados más simples. Se plantea una prueba en dos etapas, típica en la verificación de programas concurrentes. En la primera etapa, se prueban los procesos secuenciales (estas pruebas se denominan pruebas *locales*), y en la segunda etapa se valida que las pruebas locales sean libres de interferencia.
La sensatez de la regla O-CONC se prueba por inducción, mediante un lema que es la extensión concurrente del Lema de Preservación Composicional definido previamente. Primero se define y prueba el lema, y luego se prueba la sensatez de O-CONC:

Lema de Preservación Concurrente. Sean $P :: [\ ||\ P_i\ ]$ con $P_i :: S_i$, un conjunto de proof outlines secuenciales libres de interferencia de $\{p_i\}\ P_i\ \{q_i\}$, y una computación de P con configuración inicial $C_0 = <P, \sigma_0>$, tal que $\sigma_0 \models \bigwedge_{i=1,n} p_i$. Sea $S_i^{(k)}$ la denotación de la continuación sintáctica de S_i en la configuración C_k. Se cumple que para todo i, con $1 \le i \le n$, y toda configuración $C_k = <[\ ||\ S_i^{(k)}\ ], \sigma_k>$, con $k \ge 0$, si $S_i^{(k)} \neq E$, entonces $\sigma_k \models pre(S_i^{(k)})$, donde $pre(S_i^{(k)})$ es la precondición de $S_i^{(k)}$ en la proof outline secuencial i-ésima, y si $S_i^{(k)} = E$, entonces $\sigma_k \models post(P_i)$.

Prueba. Se hará inducción sobre k. Para facilitar la escritura, se asume que si $S_i^{(k)} = E$, entonces $pre(S_i^{(k)}) = post(P_i)$.

Inicialmente se cumple $\sigma_0 \models \bigwedge_{i=1,n} p_i$ por hipótesis, es decir que $\sigma_0 \models \bigwedge_{i=1,n} pre(S_i^{(0)})$.

Supóngase que, dada $C_k = <[\ ||\ S_i^{(k)}\ ], \sigma_k>$, con $k \ge 0$, se cumple $\sigma_k \models \bigwedge_{i=1,n} pre(S_i^{(k)})$. Sea la transición $C_k \rightarrow_m C_{k+1} = <[\ ||\ S_i^{(k+1)}\ ], \sigma_{k+1}>$. Por la definición de «$\rightarrow$», se cumple que para todo $i \neq m$, $S_i^{(k)} = S_i^{(k+1)}$, y por lo tanto $pre(S_i^{(k)}) = pre(S_i^{(k+1)})$. Por la sensatez de $H \cup \{AWAIT\}$, se cumple que $\sigma_{k+1} \models pre(S_m^{(k+1)})$. Y por ser las proof outlines secuenciales libres de interferencia, se cumple que para todo $i \neq m$ sigue valiendo $\sigma_{k+1} \models pre(S_i^{(k)})$, por lo que vale $\sigma_{k+1} \models pre(S_i^{(k+1)})$.

De este modo se cumple $\sigma_{k+1} \models \bigwedge_{i=1,n} pre(S_i^{(k+1)})$.

Prueba de sensatez de la regla O-CONC. Sea un conjunto de proof outlines secuenciales libres de interferencia de $\{p_i\}\ P_i\ \{q_i\}$, y una computación π de P con configuración inicial $C_0 = <P, \sigma_0>$, tal que π termina y $\sigma_0 \models \bigwedge_{i=1,n} p_i$. Sea σ' el estado final de π. Se cumple $\sigma' \models \bigwedge_{i=1,n} q_i$

por el Lema de Preservación Concurrente, dado que en la configuración final de π, es decir C' = <[|| E], σ'>, se cumple $\sigma' \models \bigwedge_{i=1,n} post(P_i)$.

□

El siguiente es un primer ejemplo de prueba de correctitud parcial utilizando los axiomas y reglas presentados:

Ejemplo. Vale |– {x = 0} P :: [await true $\rightarrow$ x := x + 1 end || await true $\rightarrow$ x := x + 2 end] {x = 3}.

Se prueba fácilmente (queda como ejercicio para el lector) que:

{x = 0 $\vee$ x = 2} {x = 0 $\vee$ x = 1}
[await true $\rightarrow$ x := x + 1 end || await true $\rightarrow$ x := x + 2 end]
{x = 1 $\vee$ x = 3} {x = 2 $\vee$ x = 3}

es una proof outline concurrente de {x = 0} P {x = 3}.

Las disyunciones expresadas en las aserciones intermedias del ejemplo son típicas en estas pruebas. Un esquema como el de la figura 3.4.1 contribuye a definirlas:

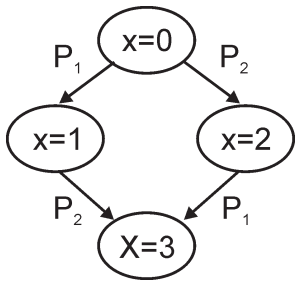


Figura 3.4.1

Por ejemplo, antes de la ejecución del primer proceso de P el valor de x puede ser 0 ó 2, antes de la ejecución del segundo proceso de P el valor de x puede ser 0 ó 1, etc.

Se hace necesario debilitar las aserciones de las proof outlines secuenciales para que resulten libres de interferencia. Esto tendrá implicancias en la completitud relativa del método, como se verá enseguida.
El siguiente es un ejemplo un poco más complejo de prueba de correctitud parcial:

Ejemplo. Con el siguiente programa P_{cfac} de SVL se pretende calcular concurrentemente el factorial de un número natural N > 1:

P_{cfac} :: c_1 := true ; c_2 :: true ; i := 1 ; k := N ; n := N ;
[while c_1 do await true $\rightarrow$ if i + 1 < k then i := i + 1 ; n := n **.** i
else c_1 := false fi end od
||
while c_2 do await true $\rightarrow$ if k – 1 > i then k := k – 1 ; n := n **.** k
else c_2 := false fi end od]

Se va a probar que P_{cfac} es parcialmente correcto con respecto a <N > 1, n = N!>. Se propone la siguiente proof outline concurrente:

{inv: i < k $\wedge$ ($\neg$ c_1 $\rightarrow$ i + 1 = k) $\wedge$ n(i + 1)…(k – 1) = N!}
[while c_1 do
{c_1 $\wedge$ i < k $\wedge$ n(i + 1)…(k – 1) = N!}
await true $\rightarrow$ if i + 1 < k then i := i + 1 ; n := n **.** i else c_1 := false fi end
{i < k $\wedge$ ($\neg$ c_1 $\rightarrow$ i + 1 = k) $\wedge$ n(i + 1)…(k – 1) = N!}
od
{i < k $\wedge$ i + 1 = k $\wedge$ n(i + 1)…(k – 1) = N!}

||

{inv: i < k $\wedge$ ($\neg$ c_2 $\rightarrow$ k – 1 = i) $\wedge$ n(i + 1)…(k – 1) = N!}
[while c_2 do
{c_2 $\wedge$ i < k $\wedge$ n(i + 1)…(k – 1) = N!}
await true $\rightarrow$ if k – 1 > i then k := k – 1 ; n := n **.** k else c_2 := false fi end
{i < k $\wedge$ ($\neg$ c_2 $\rightarrow$ k – 1 = i) $\wedge$ n(i + 1)…(k – 1) = N!}
od]
{i < k $\wedge$ k – 1 = i $\wedge$ n(i + 1)…(k – 1) = N!}

Queda como ejercicio para el lector, verificar que la proof outline concurrente propuesta es correcta, y probar que {N > 1} P_{cfac} {n = N!}.

Supóngase ahora la siguiente variante del sencillo programa P de SVL presentado anteriormente:

P' :: [await true $\rightarrow$ x := x + 1 end || await true $\rightarrow$ x := x + 1 end]

Se cumple $\models$ {x = 0} P' {x = 2}. Sin embargo, se puede demostrar formalmente que esta fórmula de correctitud no puede probarse en H $\cup$ {AWAIT, O-CONC). En base al esquema de la figura 3.4.2:

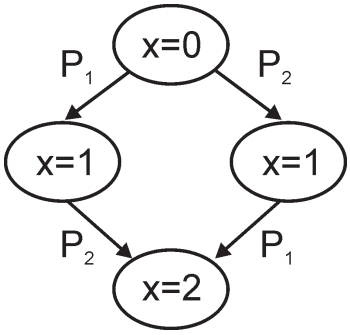


Figura 3.4.2

una proof outline concurrente candidata de {x = 0} P' {x = 2} sería:

{x = 0 $\vee$ x = 1} {x = 0 $\vee$ x = 1}
[await true $\rightarrow$ x := x + 1 end || await true $\rightarrow$ x := x + 1 end]
{x = 1 $\vee$ x = 2} {x = 1 $\vee$ x = 2}

pero como se comprueba fácilmente, no lo es. Por ejemplo, notar que no se cumple:

((x = 1 $\vee$ x = 2) $\wedge$ (x = 1 $\vee$ x = 2)) $\rightarrow$ (x = 2).

En este caso, lo que sucede es que las aserciones intermedias son demasiado débiles. A diferencia de lo que ocurría en el ejemplo anterior, ahora se progrese primero por un proceso o por el otro, el estado intermedio σ es el mismo, con $\sigma \models$ x = 1. En otras palabras, utilizando solamente la variable de programa x no alcanza para poder registrar precisamente la historia del interleaving llevado a cabo.

Para resolver esta incompletitud, se agrega una nueva regla al método, que permite ampliar el programa original con variables auxiliares para fortalecer las aserciones intermedias, pero sin afectar el cómputo básico. Por ejemplo, la prueba anterior podría mejorarse de la siguiente manera (se resalta lo que tiene que ver con las variables auxiliares):

$$\{x = 0\}\ \mathbf{y := 0}\ ;\ \mathbf{z := 0}\ ;\ \{x = 0 \wedge \mathbf{y = 0 \wedge z = 0}\}$$
$$\{(x = 0 \wedge \mathbf{y = 0 \wedge z = 0}) \vee (x = 1 \wedge \mathbf{y = 0 \wedge z = 1})\}\ \{(x = 0 \wedge \mathbf{y = 0 \wedge z = 0}) \vee (x = 1 \wedge \mathbf{y = 1 \wedge z = 0})\}$$
$$[P_1 :: \text{await true} \rightarrow x := x + 1\ ;\ \mathbf{y := 1}\ \text{end}\ \ \|\ \ P_2 :: \text{await true} \rightarrow x := x + 1\ ;\ \mathbf{z := 1}\ \text{end}]$$
$$\{(x = 1 \wedge \mathbf{y = 1 \wedge z = 0}) \vee (x = 2 \wedge \mathbf{y = 1 \wedge z = 1})\}\ \{(x = 1 \wedge \mathbf{y = 0 \wedge z = 1}) \vee (x = 2 \wedge \mathbf{y = 1 \wedge z = 1})\}$$
$$\{x = 2\}$$

Es decir, se agregan variables auxiliares y, z, más asignaciones a las mismas. Las variables auxiliares tienen inicialmente el valor 0, y cambian a 1 cuando los procesos asociados P_1 y P_2, respectivamente, ejecutan la instrucción await. Las nuevas asignaciones no alteran el cómputo básico, y las aserciones intermedias son más fuertes que antes. Todo esto permite obtener una proof outline concurrente, pero con respecto al programa P' ampliado con las variables auxiliares. Sin embargo, por el tipo de modificaciones llevadas a cabo, también se obtiene una proof outline concurrente con respecto al programa original, lo que se formaliza a continuación:

Definición. El conjunto A es un conjunto de variables auxiliares con respecto al programa P ε SVL y a las aserciones p y q de Assn, si cumple:

a) $A \cap var(q) = \varnothing$, siendo var(q) el conjunto de variables libres de la aserción q.
b) Si x ε A, entonces x aparece sólo en asignaciones de P.
c) Para toda asignación x := e de P, si existe una variable y ε A en la expresión e, entonces también x ε A.

□

Notar que no se establecen restricciones con respecto a p: puede haber variables x en $A \cap var(p)$, siendo var(p) el conjunto de variables libres de la aserción p. En este caso, las variables x se tratarían como variables de especificación.
Con estas consideraciones, se agrega al método de verificación de correctitud parcial la siguiente regla, denominada AUX (por variables auxiliares):

$$\frac{\{p\}\ P\ \{q\}}{\{p\}\ P_{|A}\ \{q\}}$$

donde $P_{|A}$ denota el programa P sin las nuevas asignaciones con variables auxiliares. Se prueba fácilmente que la regla AUX es sensata, lo que queda como ejercicio para el lector.

El método de verificación O se define finalmente como H $\cup$ {AWAIT, O-CONC, AUX).

3.4.1.3. Verificación de la correctitud total de los programas de SVL

La correctitud total de los programas de SVL considera mínimamente la correctitud parcial, la ausencia de deadlock (no hay computaciones finitas que terminan en el estado de falla) y la no divergencia (no hay computaciones infinitas). Podría tratarse además la exclusión mutua y la ausencia de inanición.
El método O* (es decir, el método O ampliado para la prueba de correctitud total de los programas de SVL) plantea pruebas separadas para cada propiedad.

Para la prueba de ausencia de deadlock se utiliza la regla O-DEADLOCK (que es una meta-regla). O-DEADLOCK establece las siguientes fases, a partir de un programa P :: [|| P_i] de SVL y una especificación <p, q>:

1) Obtener una proof outline concurrente de {p} P {q}.

2) Marcar en la proof outline concurrente todos los casos posibles de deadlock. Esto se hace identificando n-tuplas $C_i = <\lambda_1, \lambda_2, \ldots, \lambda_n>$, tantas como casos posibles de deadlock existan, tal que P tiene n procesos, y cada λ_k es una etiqueta asociada a un await o al final de un proceso. Toda C_i tiene al menos una etiqueta asociada a un await (de lo contrario no representaría un caso posible de deadlock).

3) Caracterizar semánticamente las n-tuplas C_i mediante aserciones δ_i, y probar que todas son falsas. Las aserciones δ_i se denominan *imágenes semánticas*. Son conjunciones de aserciones $\delta_{i\,k}$ que se definen de la siguiente manera: (a) Si la etiqueta λ_k de C_i está asociada a una instrucción T :: await B $\rightarrow$ S end, entonces $\delta_{i\,k}$ = pre(T) $\wedge \neg$ B. (b) Si la etiqueta λ_k de C_i está asociada al final del proceso P_k, entonces $\delta_{i\,k}$ = post(P_k).

Dado que se parte de una proof outline concurrente de {p} P {q}, las fases (2) y (3) aseguran la ausencia de deadlock en P, a partir de un estado inicial σ que satisface p.

□

Por ejemplo, sea el siguiente esquema de proof outline concurrente de {p} P {q}, en el que ya aparecen insertas las etiquetas λ:

$$
\begin{array}{l}
P :: [P_1 :: \ldots \{pre(T_1)\}\ \boldsymbol{\lambda_1} T_1 :: \text{await } B_1 \rightarrow S_1 \text{ end} \ldots\ \boldsymbol{\lambda_2}\{post(P_1)\} \\
\quad \| \\
\quad P_2 :: \ldots\ \boldsymbol{\lambda_3}\{post(P_2)\} \\
\quad \| \\
\quad P_3 :: \ldots \{pre(T_2)\}\ \boldsymbol{\lambda_4} T_2 :: \text{await } B_2 \rightarrow S_2 \text{ end} \ldots\ \boldsymbol{\lambda_5}\{post(P_3)\}]
\end{array}
$$

Los casos posibles de deadlock son:

$C_1 = <\lambda_1, \lambda_3, \lambda_4>$, $C_2 = <\lambda_1, \lambda_3, \lambda_5>$, y $C_3 = <\lambda_2, \lambda_3, \lambda_4>$.

Las imágenes semánticas asociadas a los C_i son:

$\delta_1 = (pre(T_1) \wedge \neg B_1) \wedge post(P_2) \wedge (pre(T_2) \wedge \neg B_2)$.
$\delta_2 = (pre(T_1) \wedge \neg B_1) \wedge post(P_2) \wedge post(P_3)$.
$\delta_3 = post(P_1) \wedge post(P_2) \wedge (pre(T_2) \wedge \neg B_2)$.

Finalmente, verificando que las aserciones δ_1, δ_2 y δ_3 son falsas, se prueba que P, a partir de un estado $\sigma \models p$, no tiene deadlock.
La proof outline concurrente utilizada puede ser distinta a la que se emplee para la prueba de correctitud parcial. En el siguiente ejemplo, se prueba la ausencia de deadlock en el programa presentado anteriormente para implementar el problema del productor-consumidor:

Ejemplo. Dado el programa P_{pc}, y la precondición $p = (in = out = 0 \wedge i = k = 1 \wedge b > 0 \wedge n > 0)$, se prueba la ausencia de deadlock en P_{pc}. Se propone la siguiente proof outline concurrente, en la que ya se marcan las etiquetas asociadas a los posibles casos de deadlock:

$\{inv: 0 \leq in - out \leq b \wedge 1 \leq i \leq n + 1 \wedge 1 \leq k \leq n + 1 \wedge b > 0 \wedge\ i = in + 1\}$
[while $i \leq n$ do
$\{0 \leq in - out \leq b \wedge 1 \leq i \leq n \wedge 1 \leq k \leq n + 1 \wedge b > 0 \wedge i = in + 1\}$
    x := A[i] ;

$\{0 \le in - out \le b \wedge 1 \le i \le n \wedge 1 \le k \le n + 1 \wedge b > 0 \wedge i = in + 1\}$
$\lambda_1 \rightarrow$ await $in - out < b \rightarrow$ skip end ;
$\{0 \le in - out < b \wedge 1 \le i \le n \wedge 1 \le k \le n + 1 \wedge b > 0 \wedge i = in + 1\}$
buffer[in mod b] := x ;
$\{0 \le in - out < b \wedge 1 \le i \le n \wedge 1 \le k \le n + 1 \wedge b > 0 \wedge i = in + 1\}$
in := in + 1 ;
$\{0 \le in - out \le b \wedge 1 \le i \le n \wedge 1 \le k \le n + 1 \wedge b > 0 \wedge i = in\}$
i := i + 1
$\{0 \le in - out \le b \wedge 1 \le i \le n + 1 \wedge 1 \le k \le n + 1 \wedge b > 0 \wedge i = in + 1\}$
od
$\{0 \le in - out \le b \wedge i = n + 1 \wedge 1 \le k \le n + 1 \wedge b > 0 \wedge i = in + 1\}$
$\lambda_2 \rightarrow$

||

$\{inv: 0 \le in - out \le b \wedge 1 \le i \le n + 1 \wedge 1 \le k \le n + 1 \wedge b > 0 \wedge k = out + 1\}$
while $k \le n$ do
$\{0 \le in - out \le b \wedge 1 \le i \le n + 1 \wedge 1 \le k \le n \wedge b > 0 \wedge k = out + 1\}$
$\lambda_3 \rightarrow$ await $in - out > 0 \rightarrow$ skip end ;
$\{0 < in - out \le b \wedge 1 \le i \le n + 1 \wedge 1 \le k \le n \wedge b > 0 \wedge k = out + 1\}$
y := buffer[out mod b] ;
$\{0 < in - out \le b \wedge 1 \le i \le n + 1 \wedge 1 \le k \le n \wedge b > 0 \wedge k = out + 1\}$
out := out + 1 ;
$\{0 \le in - out \le b \wedge 1 \le i \le n + 1 \wedge 1 \le k \le n \wedge b > 0 \wedge k = out\}$
B[k] := y ;
$\{0 \le in - out \le b \wedge 1 \le i \le n + 1 \wedge 1 \le k \le n \wedge b > 0 \wedge k = out\}$
k := k + 1
$\{0 \le in - out \le b \wedge 1 \le i \le n + 1 \wedge 1 \le k \le n + 1 \wedge b > 0 \wedge k = out + 1\}$
od]
$\{0 \le in - out \le b \wedge 1 \le i \le n + 1 \wedge k = n + 1 \wedge b > 0 \wedge k = out + 1\}$
$\lambda_4 \rightarrow$

Queda como ejercicio para el lector, verificar que la proof outline concurrente propuesta es correcta, y completar la prueba.

Para verificar la no divergencia se utiliza la (meta-)regla O-CONC*, que al igual que O-CONC establece una prueba en dos etapas:

1) Obtener una proof outline secuencial de correctitud total de $<p_i> S_i <q_i>$, para todo i.

2) Verificar que las proof outlines son libres de interferencia, y además que para todo invariante parametrizado p(w) normal utilizado en la prueba de no divergencia de S_i, y para toda instrucción S' de S_k, con $k \neq i$ y siendo S' un await o una asignación normal, entonces se cumple la fórmula $<p(w) \wedge pre(S')>\ S'\ <\exists v \leq w : p(v)>$.

□

Es decir, no basta con probar aisladamente en cada proceso que todo while termina, sino que también debe verificarse que ninguna instrucción de otro proceso que puede modificar variables (await o asignaciones normales), perjudique la terminación del while incrementando el variante asociado (podría en cambio acelerar su terminación si lo decrementara). Como en el caso de la prueba de ausencia de deadlock, la proof outline concurrente puede ser distinta a las utilizadas para las pruebas de las otras propiedades.
En el siguiente ejemplo, se presenta una prueba de no divergencia del programa para el problema del productor-consumidor:

Ejemplo. Sea otra vez el programa P_{pc} y la precondición ($in = out = 0 \wedge i = k = 1 \wedge b > 0 \wedge n > 0$). Se prueba que P_{pc} no diverge. Se propone la siguiente proof outline concurrente:

<inv: 1 ≤ i ≤ n + 1, fc: w = n + 1 – i>
[while i ≤ n do
<1 ≤ i ≤ n>
x := A[i] ;
<1 ≤ i ≤ n>
await in – out < b → skip end ;
<1 ≤ i ≤ n>
buffer[in mod b] := x ;
<1 ≤ i ≤ n>
in := in + 1 ;
<1 ≤ i ≤ n>
i := i + 1
<1 ≤ i ≤ n + 1>
od
<1 ≤ i ≤ n + 1>
||
<inv: 1 ≤ k ≤ n + 1, fc: v = n + 1 – k>
while k ≤ n do

<1 ≤ k ≤ n>
await in – out > 0 → skip end ;
<1 ≤ k ≤ n>
y := buffer[out mod b] ;
<1 ≤ k ≤ n>
out := out + 1 ;
<1 ≤ k ≤ n>
B[k] := y ;
<1 ≤ k ≤ n>
k := k + 1
<1 ≤ k ≤ n + 1>
od]
<1 ≤ k ≤ n + 1>

Queda como ejercicio para el lector, verificar que la proof outline concurrente propuesta es correcta, y completar la prueba.

Asumiendo fairness, la prueba de no divergencia contempla la posibilidad de instrucciones while que, consideradas aisladamente en un proceso, pueden no terminar. Sea, por ejemplo, el programa de SVL siguiente:

P :: [while x = 1 do skip od || x := 0].

Notar que si el fairness asegura que la asignación x := 0 se va a ejecutar alguna vez, entonces el programa P termina aún cuando el while, considerado aisladamente, diverge.

La prueba de exclusión mutua es similar a la prueba de ausencia de deadlock. Ambas propiedades, al igual que la correctitud parcial, son del tipo safety, y por lo tanto se prueban inductivamente.

La prueba de ausencia de inanición es similar a la prueba de no divergencia. Ambas propiedades son del tipo liveness, y por lo tanto se prueban mediante variantes pertenecientes a conjuntos parcialmente ordenados bien fundados.

Antes de pasar a tratar la verificación de los programas distribuidos, en las próximas dos secciones se consideran programas concurrentes con otra primitiva de sincronización, que permite desarrollar pruebas más estructuradas que las estudiadas utilizando el lenguaje SVL.

3.4.1.4. Lenguaje RVL

El lenguaje RVL (por *Resource Variables Language* o lenguaje de recursos de variables) tiene una primitiva de sincronización que fuerza la exclusión mutua entre las secciones de programa que modifican variables compartidas. Las variables compartidas se disponen en conjuntos de variables que se denominan *recursos*, sobre los que los procesos obtienen, al utilizarlos, acceso exclusivo. Esta característica del lenguaje permite que las pruebas de correctitud de los programas sean más estructuradas que las que se vieron en las secciones precedentes.
Un programa concurrente P ε RVL tiene la forma:

P :: resource r_1 (var-list$_1$) ; … ; resource r_m (var-list$_m$) ; P_0 ; [P_1 || P_2 || … || P_n].

Los recursos r_k , con $1 \leq k \leq m$, son conjuntos disjuntos dos a dos de variables, enumeradas en las listas respectivas var-list$_k$, sobre los que los procesos secuenciales P_i :: S_i, con $1 \leq i \leq n$, obtienen acceso exclusivo al utilizarlos.
El subprograma P_0 pertenece al lenguaje PLW, y está compuesto por asignaciones de inicialización de variables. En cambio, las instrucciones de los procesos pueden ser de PLW o bien la instrucción de sincronización with, cuya sintaxis es:

with r_k when B do S endwith,

siendo r_k algún recurso, B una guardia booleana, y S un subprograma de PLW. Para simplificar la presentación, no se permite el anidamiento de concurrencia ni de las instrucciones with.
Las variables de los recursos sólo pueden ser utilizadas por las sentencias with. Además, toda variable compartida modificable por un proceso, debe estar definida dentro de un recurso.
Informalmente, cuando un proceso P_i ejecuta una instrucción with r_k when B do S endwith, si el recurso r_k no está siendo utilizado por otro proceso y la guardia B es verdadera, P_i puede ocupar el recurso, utilizar sus variables y progresar en su ejecución. El resto de los procesos no tiene acceso al recurso r_k mientras P_i no lo libere, lo que ocurre recién cuando P_i termina de ejecutar la instrucción with.
Notar que en un programa de RVL se pueden, entonces, identificar sintácticamente las secciones con variables compartidas modificables. Dichas secciones reciben el nombre de *secciones críticas condicionales*.

Formalmente, la semántica operacional de RVL se define de la siguiente manera (sólo se describe lo nuevo con respecto a lo que se presentó en las secciones anteriores):

Sea $\rho[k]$, $1 \leq k \leq m$, un arreglo asociado a los m recursos r_k de un programa P de RVL, tal que $\rho[k] = 0$ ó 1, según r_k esté libre u ocupado, respectivamente. Al inicio se cumple que para todo k, $\rho[k] = 0$. Se define:

(1) Si $S_i \neq$ with y $\langle S_i , \sigma\rangle \rightarrow \langle S'_i , \sigma'\rangle$, entonces:

$$\langle[S_1 \,||\, \ldots \,||\, S_i \,||\, \ldots \,||\, S_n], \rho, \sigma\rangle \rightarrow_{\langle i, \sigma\rangle} \langle[S_1 \,||\, \ldots \,||\, S'_i \,||\, \ldots \,||\, S_n], \rho, \sigma'\rangle.$$

Es decir, las instrucciones que no son with no modifican la situación de los recursos.

(2) Si $\rho[k] = 0$ y $\sigma \models B$, entonces:

$\langle[S_1 \,||\, \ldots \,||$ with r_k when B do S_i endwith$_k \,||\, \ldots \,||\, S_n], \rho, \sigma\rangle \rightarrow_{\langle i, \sigma\rangle}$
$\langle[S_1 \,||\, \ldots \,||\, S_i$ endwith$_k \,||\, \ldots \,||\, S_n], \rho', \sigma\rangle$, con $\rho'[k] = 1$ y $\rho'[h] = \rho[h]$ para todo $h \neq k$.

El subíndice k en endwith$_k$ no forma parte de la sintaxis de RVL, sólo se utiliza para facilitar la definición. El chequeo de ocupación de un recurso y la evaluación de una guardia booleana no modifican el estado corriente. Notar que a diferencia de la instrucción await, el with no es atómico.

(3) $\langle[S_1 \,||\, \ldots \,||$ endwith$_k \,||\, \ldots \,||\, S_n], \rho, \sigma\rangle \rightarrow_{\langle i, \sigma\rangle}$
$\langle[S_1 \,||\, \ldots \,||\, E \,||\, \ldots \,||\, S_n], \rho', \sigma\rangle$, con $\rho'[k] = 0$ y $\rho'[h] = \rho[h]$ para todo $h \neq k$.

□

Por lo tanto, en ninguna computación de un programa de RVL puede existir una configuración concurrente de la forma $\langle[S_1 \,||\, \ldots \,||\, S_i \,||\, \ldots \,||\, S_k \,||\, \ldots \,||\, S_n], \rho, \sigma\rangle$, con $i \neq k$, tal que las continuaciones sintácticas S_i y S_k comiencen con una instrucción interna de un with asociado a un mismo recurso (la prueba queda como ejercicio para el lector).
La ejecución de la instrucción with puede causar deadlock, y los while siguen siendo la única causa de divergencia.
Para simplificar la presentación, no se asume ninguna hipótesis de fairness. La única asunción en este sentido es que mientras un programa puede progresar en alguno de sus procesos entonces progresa, lo que se conoce como *propiedad de liveness fundamental*.

Por otro lado, se asume que no existe ninguna política de priorización sobre la ocupación de los recursos. La idea es que toda vez que dos o más procesos compiten por un recurso, su ocupación se decide no determinísticamente. Podría trabajarse con una primitiva de sincronización más sofisticada, que maneje prioridades sobre el acceso a los recursos, como el *monitor*. El monitor permite establecer políticas de priorización entre los procesos, y también la liberación temporaria de los recursos (además se pueden utilizar para implementar *tipos de datos abstractos*, al permitir encapsular datos y definir nuevas operaciones sobre los mismos). Hay un ejercicio al final de esta parte del libro referido a la semántica de RVL con monitores.

El siguiente ejemplo presenta un (esquema de) programa de RVL, relacionado con un problema muy conocido:

Ejemplo. El siguiente (esquema de) programa P_{fcp} implementa una solución concurrente para el clásico problema de los filósofos que comen y piensan. Cinco filósofos están sentados alrededor de una mesa circular y alternativamente piensan y comen espagueti. Los filósofos necesitan dos tenedores para comer, hay sólo cinco tenedores sobre la mesa, cada uno en medio de dos filósofos, y los únicos tenedores que puede utilizar un filósofo son los que están a su derecha e izquierda. De esta manera dos filósofos vecinos no pueden comer al mismo tiempo:

P_{fpc} :: resource forks(f);
f[0]:= 2 ; f[1] := 2 ; f[2] := 2 ; f[3] := 2 ; f[4] := 2 ;
[DP_0 || DP_1 || DP_2 || DP_3 || DP_4], tal que, para $0 \leq i \leq 4$:

DP_i :: while true do
with forks when f[i] = 2 do
f[i – 1] := f[i – 1] – 1 ; f[i + 1] := f[i + 1] – 1
endwith;
… sección «$comer_i$» …
with forks when true do
f[i – 1] := f[i – 1] + 1 ; f[i + 1] := f[i + 1] + 1
endwith;
… sección «$pensar_i$» …
od

El proceso DP_i representa al filósofo i. El arreglo f almacena el número de tenedores disponibles de cada uno de los filósofos en todo momento

(la aritmética sobre los índices es módulo 5). Las secciones «comer$_i$» y «pensar$_i$» no modifican el arreglo f.

3.4.1.5. Verificación de la correctitud parcial y total de los programas de RVL

Para verificar la correctitud parcial de los programas de RVL, se utilizan los axiomas y reglas SKIP, ASI, SEC, COND, REP, CONS y AUX definidos anteriormente, más dos nuevas reglas, asociadas al with y la composición concurrente. El método de verificación se denomina R.

La regla asociada a la instrucción with (WITH) se define de la siguiente manera:

$$\frac{\{I_r \wedge p \wedge B\}\ S\ \{I_r \wedge q\}}{\{p\}\ \text{with r when B do S endwith}\ \{q\}}$$

Para simplificar la presentación, de ahora en más se asumirá la existencia de un solo recurso r. La aserción I_r es un invariante asociado al recurso r, que vale toda vez que el recurso está libre. Las variables de I_r están en r.
La regla WITH establece que si se cumple $\{p \wedge B\}\ S\ \{q\}$, entonces también se cumple {p} with r when B do S endwith {q}, pero con la condición de que la ejecución de S preserve, a partir de $p \wedge B$, el invariante I_r . La utilidad de esta regla se entenderá tras la presentación de la siguiente.

□

La (meta-)regla asociada a la composición concurrente en RVL, denominada R-CONC, es:

$$\frac{\{p_i\}\ S^*_i\ \{q_i\},\ 1 \le i \le n,\ \text{son proof outlines secuenciales que utilizan } I_r}{\{I_r \wedge \bigwedge_{i=1,n} p_i\}\ [\ ||_{i=1,n}\ S_i\ ]\ \{I_r \wedge \bigwedge_{i=1,n} q_i\}}$$

tal que:

a) Para todo i, para todo k, si $i \neq k$, entonces las variables libres de las aserciones de $\{p_i\}\ S^*_i\ \{q_i\}$ son disjuntas con las variables modificables de P_k.
b) Las variables libres del invariante I_r están en el recurso r.

□

R-CONC establece, otra vez, una verificación en dos etapas:

La primera etapa consiste en desarrollar pruebas locales para los procesos secuenciales, las cuales se basan en el invariante I_r. El caso S :: with r when B do S' endwith en una proof outline secuencial se define de la siguiente manera:

(a) $pre(S) \wedge B \rightarrow pre(S')$, y (b) $post(S') \rightarrow post(S)$.

En la segunda etapa se obtiene una proof outline concurrente a partir de las proof outlines secuenciales. Ahora no se necesita chequear que las pruebas locales sean libres de interferencia, porque sus aserciones se refieren únicamente a variables locales o a variables compartidas de solo lectura. La información de las variables compartidas modificables se propaga desde el comienzo hasta el final de la proof outline concurrente a través del invariante I_r, por medio de las distintas aplicaciones de la regla WITH. Al comienzo vale I_r porque el recurso r está libre. De este modo, las pruebas de los programas RVL son más estructuradas que las de los programas de SVL.

Ejemplo. El siguiente programa de RVL implementa la exclusión mutua entre dos secciones críticas:

```
P :: resource semaforo(s) ;
    [P1 :: with semaforo when s = 1 do s := 0 endwith ;
           … sección crítica 1 …
        with semaforo when true do s := 1 endwith
    ||
     P2 :: with semaforo when s = 1 do s := 0 endwith ;
           … sección crítica 2 …
           with semaforo when true do s := 1 endwith]
```

Se cumple $|-_R \{s = 1\}\ P\ \{s = 1\}$:

Se van a utilizar variables auxiliares a y b que, respectivamente, valdrán 1 cuando P_1 y P_2 estén en sus secciones críticas, y 0 en caso contrario. La siguiente es una proof outline concurrente de {s = 1} P' {s = 1}, siendo P' el programa ampliado con las variables auxiliares, y el invariante asociado al recurso semáforo:

$I_{sem} = [(s = 0) \rightarrow (a = 1 \wedge b = 0) \vee (a = 0 \wedge b = 1)] \wedge [(s = 1) \rightarrow (a = 0 \wedge b = 0)] \wedge [0 \leq s \leq 1]$.

```
P' ::   resource semaforo(s, a, b) ; a := 0 ; b := 0 ;
        {a = 0}
        [P1 :: with semaforo when s = 1 do s := 0 ; a := 1 endwith ; {a = 1}
                … sección crítica 1 … ; {a = 1}
               with semaforo when true do s := 1 ; a := 0 endwith
        {a = 0}
        ||
        {b = 0}
         P2 :: with semaforo when s = 1 do s := 0 ; b := 1 endwith ; {b = 1}
                … sección crítica 2 … ; {b = 1}
               with semaforo when true do s := 1 ; b := 0 endwith]
        {b = 0}
```

Se cumple $(s = 1 \wedge a = 0 \wedge b = 0) \rightarrow (I_{sem} \wedge a = 0 \wedge b = 0)$ y $(I_{sem} \wedge a = 0 \wedge b = 0) \rightarrow (s = 1)$. Por lo tanto, empleando CONS y AUX, vale: {s = 1} P {s = 1}.

La prueba de ausencia de deadlock en RVL es muy similar a la que se describió para SVL. La (meta-)regla R-DEADLOCK establece, a partir de una proof outline concurrente de {p} P {q}, una fase sintáctica de identificación de todos los casos posibles C_i de deadlock, y luego una fase semántica de caracterización de dichos casos mediante imágenes semánticas δ_i y la prueba de que todas las δ_i son falsas.

Ahora, las imágenes semánticas $\delta_i = \bigwedge_i \delta_{i\,k}$ se definen de la siguiente manera:

- Si la etiqueta λ_k de C_i precede a una instrucción T :: with r when B do S endwith, entonces la parte correspondiente de la imagen semántica es $\delta_{i\,k} = pre(T) \wedge \neg B \wedge I_r$, es decir, se incorpora la información del recurso, que en un caso posible de deadlock está libre.

– Si la etiqueta λ_k de C_i está al final de un proceso P_k, entonces $\delta_{i\,k}$ = post(P_k).

□

La no divergencia también se demuestra de una manera similar a la que se presentó antes. La (meta-)regla que se utiliza es R-CONC*, que establece elaborar en una primera etapa proof outlines secuenciales de $<p_i>\ S_i\ <q_i>$, con $1 \leq i \leq n$, y en una segunda etapa integrarlas en una proof outline concurrente de <p> P <q>.
La segunda etapa es más simple que la descripta para los programas de SVL. Notar que, para todo while de un proceso P_i, o bien el variante asociado se expresa en términos de variables locales o de solo lectura, o bien el while está incluido en una sección crítica condicional sobre la que RVL asegura exclusión mutua, lo que evita que algún proceso P_k, con $i \neq k$, pueda alterar el decremento del variante y así perjudicar la terminación del while.

□

Las pruebas de exclusión mutua y ausencia de inanición en el método R* (el método R ampliado para la prueba de correctitud total de los programas RVL) son similares, respectivamente, a las de ausencia de deadlock y no divergencia.

3.4.2. Verificación de programas distribuidos

3.4.2.1. Lenguaje CSP

El lenguaje que se va a utilizar para describir la verificación de los programas distribuidos es un fragmento del lenguaje CSP (por *Communicating Sequential Processes* o procesos secuenciales que se comunican).
En el modelo distribuido, los procesos de un programa no comparten variables, y se comunican y sincronizan a través de pasajes de mensajes. CSP implementa la variante *sincrónica* del modelo distribuido (también conocida como *handshaking* o *rendezvous*), según la cual el envío de un mensaje por parte de un proceso y la recepción de ese mensaje por parte de otro proceso, coinciden en el tiempo. De esta manera, un proceso puede quedar bloqueado a la espera de que otro esté preparado para enviar o recibir un mensaje, según el caso. Al final se hará alguna

breve mención a la variante asincrónica, según la cual sólo el proceso que recibe un mensaje puede quedar bloqueado.

Un programa de CSP tiene la forma:

$$P :: [P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n].$$

Se podrán usar también las expresiones $P :: [\,||_{\,i=1,n} P_i\,]$ y $P :: [\,||\, P_i\,]$ para denotar un programa de CSP. Cada proceso se define como $P_i :: S_i$. No hay anidamiento de concurrencia. Las instrucciones de S_i son el skip, la asignación, la secuencia, y además:

(1) La instrucción de lectura P_i ? x:

Su semántica informal es: dado un proceso P_k con una instrucción P_i ? x, siendo $k \neq i$, cuando P_k está por ejecutar la lectura, si P_i está por ejecutar la escritura respectiva, que se describirá enseguida, entonces P_k puede recibir en su variable local entera x el valor enviado por P_i.

□

(2) La instrucción de escritura P_i ! e:

Su semántica informal es: dado un proceso P_k con una instrucción P_i ! e, siendo $k \neq i$, cuando P_k está por ejecutar la escritura, si P_i está por ejecutar la lectura respectiva P_k ? x, entonces P_k puede enviarle a P_i un valor que es la evaluación de la expresión entera e en su estado local corriente.

La lectura y la escritura son instrucciones atómicas. Se conocen genéricamente como instrucciones de comunicación o de entrada/salida. Se dice que la instrucción P_i ? x del proceso P_k y la instrucción P_k ! e del proceso P_i *«matchean» sintácticamente* (o se acoplan sintácticamente). Ambas instrucciones, en conjunto, permiten comunicar y sincronizar procesos secuenciales. La escritura y la lectura asociada se esperan una a la otra para ejecutarse al mismo tiempo, y por lo tanto pueden causar deadlock.

□

(3) La instrucción de selección condicional no determinística:

Es igual que la del lenguaje GCL, pero ahora las guardias pueden incluir una instrucción de comunicación. La forma más general de esta instrucción es:

$$[B_1 ; \alpha_1 \rightarrow S_1 \square \ldots \square B_n ; \alpha_n \rightarrow S_n],$$

que se puede denotar $[\square_i B_i ; \alpha_i \rightarrow S_i]$, con $i \varepsilon \Delta$. Cada α_i es una instrucción de lectura o escritura. Las guardias pueden tener una expresión booleana, o bien una instrucción de comunicación, o bien la secuencia de ámbas y en ese orden. La evaluación de las guardias es no determinística:

- Se determina cuáles expresiones booleanas son verdaderas y cuáles comunicaciones pueden realizarse.
- Si existen direcciones con estas características, el proceso puede progresar por una de ellas ejecutando el subprograma S_i respectivo, y al terminar, continúa con la instrucción siguiente a la selección condicional.
- Si en cambio todas las expresiones booleanas son falsas, la selección condicional falla.
- En el caso restante, la selección condicional queda bloqueada, temporaria o definitivamente.

□

(4) La instrucción de repetición no determinística:

Es igual que la del lenguaje GCL, pero como en el caso anterior, las guardias pueden incluir una instrucción de comunicación. La forma más general de esta instrucción es:

$$*[B_1 ; \alpha_1 \rightarrow S_1 \square \ldots \square B_n ; \alpha_n \rightarrow S_n],$$

que se puede denotar $*[\square_i B_i ; \alpha_i \rightarrow S_i]$, con $i \varepsilon \Delta$, siendo α_i una lectura o una escritura. Puede haber guardias con una expresión booleana, con una instrucción de comunicación, o bien con la secuencia de las dos en este orden. El comportamiento de la repetición es igual al de la selección condicional, salvo que después de la ejecución de un subprograma S_i, si termina, se itera la repetición completa, hasta que eventualmente todas las expresiones booleanas resulten falsas, en cuyo caso el proceso continúa con la instrucción siguiente a la repetición.

□

Formalmente, la semánica operacional de CSP se define de la siguiente manera (se presenta sólo lo que tiene que ver con las configuraciones distribuidas):

1) $\langle[S_1 \,||\, \ldots \,||\, S_i \,||\, \ldots \,||\, S_n], \sigma\rangle \rightarrow_{\langle i, \sigma\rangle} \langle[S_1 \,||\, \ldots \,||\, E \,||\, \ldots \,||\, S_n], \sigma\rangle$,

si S_i :: skip, o bien S_i :: *[$\Box_k$ B_k ; $\alpha_k \rightarrow T_k$] y todas las expresiones booleanas B_k son falsas.

2) $\langle[S_1 \,||\, \ldots \,||\, S_i \,||\, \ldots \,||\, S_n], \sigma\rangle \rightarrow_{\langle i, \sigma\rangle} \langle[S_1 \,||\, \ldots \,||\, E \,||\, \ldots \,||\, S_n], \sigma[x \mid e]\rangle$,

si S_i :: x := e.

3) $\langle[S_1 \,||\, \ldots \,||\, S_i \,||\, \ldots \,||\, S_k \,||\, \ldots \,||\, S_n], \sigma\rangle \rightarrow_{\langle i, k, \sigma\rangle} \langle[S_1 \,||\, \ldots \,||\, S'_i \,||\, \ldots \,||\, S'_k \,||\, \ldots \,||\, S_n], \sigma'\rangle$, si

   (a) S_i :: P_k ! e, y S'_i :: E, o bien
   (b) S_i :: [$\Box_m$ B_m ; $\alpha_m \rightarrow T_m$], alguna B_m es verdadera, α_m :: P_k ! e, y S'_i :: T_m , o bien
   (c) S_i :: *[$\Box_m$ B_m ; $\alpha_m \rightarrow T_m$], alguna B_m es verdadera, α_m :: P_k ! e, y S'_i :: T_m ; S_i ;

   (d), (e) y (f) son como (a), (b) y (c), respectivamente, cambiando S_i por S_k y P_k ! e por P_i ? x;

   y para cualquiera de las combinaciones a-d, a-e, a-f, b-d, b-e, b-f, c-d, c-e, c-f, se cumple que $\sigma' = \sigma[x \mid e]$.

   Es decir que una comunicación entre dos procesos se comporta como una asignación. Notar que la evaluación de las guardias se efectúa atómicamente, no así la ejecución del comando con guardia completo.

4) Si $\langle[S_1 \,||\, \ldots \,||\, S_i \,||\, \ldots \,||\, S_n], \sigma\rangle \rightarrow_{\langle i, \sigma\rangle} \langle[S_1 \,||\, \ldots \,||\, S'_i \,||\, \ldots \,||\, S_n], \sigma'\rangle$, entonces

   $\langle[S_1 \,||\, \ldots \,||\, S_i ; T_i \,||\, \ldots \,||\, S_n], \sigma\rangle \rightarrow_{\langle i, \sigma\rangle} \langle[S_1 \,||\, \ldots \,||\, S'_i ; T_i \,||\, \ldots \,||\, S_n], \sigma'\rangle$ para todo T_i .

5) Si $\langle S_1 \,||\, \ldots \,||\, S_i \,||\, \ldots \,||\, S_k \,||\, \ldots \,||\, S_n], \sigma\rangle \rightarrow_{\langle i, k, \sigma\rangle} \langle[S_1 \,||\, \ldots \,||\, S'_i \,||\, \ldots \,||\, S'_k \,||\, \ldots \,||\, S_n], \sigma'\rangle$, entonces

   $\langle S_1 \,||\, .. \,||\, S_i ; T_i \,||\, .. \,||\, S_k ; T_k \,||\, .. \,||\, S_n], \sigma\rangle \rightarrow_{\langle i, k, \sigma\rangle} \langle[S_1 \,||\, .. \,||\, S'_i ; T_i \,||\, .. \,||\, S'_k ; T_k \,||\, .. \,||\, S_n], \sigma'\rangle$ para todo T_i , para todo T_k .

□

No existe un estado global único σ. En la definición anterior, σ debe entenderse como la unión de los estados asociados a cada uno de los procesos secuenciales P_i, que cuando sea necesario se denotarán con la expresión σ/i.

La función semántica asociada al lenguaje CSP se define de la siguiente manera:

$$\mathcal{M} : \text{CSP} \rightarrow (\Sigma \rightarrow \mathcal{P}(\Sigma)),$$

con $\mathcal{M}(P)(\sigma) = \{val(\pi(P, \sigma)) \mid \pi(P, \sigma) \in \Pi(P, \sigma)\}$, siendo $\Pi(P, \sigma)$ el conjunto de todas las computaciones $\pi(P, \sigma)$. Algunas computaciones pueden terminar, otras fallar y otras diverger, por lo que $\mathcal{M}(P)(\sigma)$ puede incluir el estado de falla f y el estado indefinido $\perp$. Una falla, en este caso, puede ser deadlock o una selección condicional con todas las expresiones booleanas de sus guardias falsas.

Para simplificar la presentación, se asume que no hay fairness. La única hipótesis de progreso de un programa distribuido sigue siendo la propiedad de liveness fundamental.

El siguiente ejemplo presenta algunos programas de CSP:

Ejemplo. Sea primero el programa $P_t :: [P_1 :: [P_2 ? x \rightarrow skip \,\square\, P_2 ! 0 \rightarrow skip] \,||\, P_2 :: P_1 ! 0]$.
El programa P_t termina. Hay una sola computación posible, y el valor final de x es 0.

Sea $P_f :: [P_1 :: [true \rightarrow P_2 ? x \,\square\, true \rightarrow P_2 ! 0] \,||\, P_2 :: P_1 ! 0]$.
El programa P_f puede fallar por deadlock. Hay dos computaciones posibles, y si P_f termina, entonces el valor final de x es 0.

En el siguiente programa se propaga el valor de y_1 por las variables y_2, y_3, etc., hasta llegar a la variable y_n.
$P_{prop} :: [P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n]$, con:
$P_1 :: P_2 ! y_1$
$P_i :: P_{i-1} ? y_i \,;\, P_{i+1} ! y_i$, siendo $1 < i < n$
$P_n :: P_{n-1} ? y_n$

Finalmente, el siguiente programa calcula en la variable local min del proceso U, el mínimo de los valores $a_1, a_2, \ldots, a_n$, que son los contenidos iniciales de las variables locales mi_min_i de los procesos P_i:

P_{min} :: $[P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n \,||\, U]$, con:
P_i :: $(mi\_min_i , mi\_tamaño_i) := (a_i , 1)$;
*[$\Box_{k \neq i}$ $0 < mi\_tamaño_i < n$; P_k ! $(mi\_min_i , mi\_tamaño_i) \rightarrow mi\_tamaño_i := 0$
$\Box_{k \neq i}$ $0 < mi\_tamaño_i < n$; P_k ? $(su\_min_i , su\_tamaño_i) \rightarrow$
$(mi\_min_i , mi\_tamaño_i) := (min(mi\_min_i , su\_min_i), mi\_tamaño_i + su\_tamaño_i)$] ;
[$mi\_tamaño_i = 0 \rightarrow$ skip
$\Box$
$mi\_tamaño_i = n \rightarrow U$! mi_min_i]
U :: [$\Box_{i=1,n}$ P_i ? min $\rightarrow$ skip]

La instrucción de la forma $(x_1, x_2) := (e_1, e_2)$ debe interpretarse como una asignación paralela. La función $min(x_1, x_2)$ devuelve el mínimo de los valores de las variables x_1 y x_2.

3.4.2.2. Verificación de la correctitud parcial de los programas de CSP

La verificación de los programas distribuidos se basa en las mismas ideas generales presentadas previamente.
Las pruebas se desarrollan en dos etapas. En este caso, la segunda etapa consiste en un test de cooperación sobre las puebas locales elaboradas en la primera etapa.
Se efectúan pruebas separadas para cada propiedad (correctitud parcial, ausencia de deadlock, no divergencia, etc). Las proof outlines concurrentes (que en este caso se denominan proof outlines distribuidas) pueden ser distintas en cada prueba.
Otra característica para destacar es la utilización de variables auxiliares y de invariantes globales. Un ejemplo de invariante global ya se vio en el marco de los programas de RVL, cuando se definieron invariantes asociados a los recursos. Las variables auxiliares y los invariantes globales son necesarios para que los métodos de verificación de programas distribuidos sean relativamente completos.
Se presenta a continuación el método AFR para la verificación de la correctitud parcial de los programas de CSP. AFR plantea una prueba

en dos etapas. En la primera etapa se elaboran las pruebas locales, por medio de los axiomas y reglas SKIP, ASI, SEC, CONS, y además:

(1) Axioma de lectura o de input (IN):

$$\{p\}\ P_i\ ?\ x\ \{q\},$$

con $1 \leq i \leq n$. Las aserciones p y q se refieren a variables locales del proceso P_k que lee un valor desde P_i.
El axioma establece que si $\sigma/k \models p$, entonces después de la comunicación entre P_k y P_i se cumple que el nuevo estado $\sigma'/k \models q$.
En la prueba local del proceso P_k la postcondición q debe asumirse, dado que el valor recibido por la variable x proviene de otro proceso. En la segunda etapa, cuando se considere el programa distribuido completo, se van a chequear todas las asunciones de este tipo.

□

(2) Axioma de escritura o de output (OUT):

$$\{p\}\ P_i\ !\ e\ \{p\},$$

con $1 \leq i \leq n$. La aserción p se refiere a variables locales del proceso P_k que le envía un valor a P_i.
El axioma establece que si $\sigma/k \models p$, entonces después de la comunicación entre P_k y P_i se cumple que el nuevo estado $\sigma'/k = \sigma/k \models p$.
Obviamente, la instrucción de escritura no tiene efecto sobre las variables locales.

□

(3) Regla de la selección condicional no determinística (AFR-COND):

$$\frac{\{p \wedge B_i\}\ \alpha_i\ ;\ S_i\ \{q\}}{\{p\}\ [\ \Box_i\ B_i\ ;\ \alpha_i \rightarrow S_i\ ]\ \{q\}}$$

con $i\ \varepsilon\ \Delta$. Es decir, es la misma regla que la de la selección condicional no determinística del método D para los programas de GCL, con el agregado, en el caso más general, de instrucciones de comunicación en las guardias de los comandos con guardia.

□

(4) Regla de la repetición no determinística (AFR-REP):

$$\frac{\{p \wedge B_i\}\ \alpha_i\ ;\ S_i\ \{p\}}{\{p\}\ *[\ \Box_i\ B_i\ ;\ \alpha_i \rightarrow S_i\ ]\ \{p \wedge \bigwedge_i \neg B_i\}}$$

con $i \in \Delta$. Es decir, es la misma regla que la de la repetición no determinística del método D, con el agregado, en el caso más general, de instrucciones de comunicación en las guardias de los comandos con guardia. La aserción p es el invariante de la repetición.

□

Los siguientes son los axiomas y reglas para la segunda etapa de la prueba, en que se consisten las pruebas locales y se obtiene una proof outline distribuida:

(5) Axioma de comunicación (COM):

$$\{p^x_e\}\ P_i\ ?\ x\ ||\ P_k\ !\ e\ \{p\},$$

con $1 \leq i \leq n$, $1 \leq k \leq n$, $i \neq k$. El axioma establece que el efecto de una comunicación entre dos procesos es el mismo que el de una asignación. Su utilidad se entenderá después de la presentación de la siguiente regla.

□

(6) Regla o test de cooperación (COOP):

Para todo par de instrucciones de lectura y escritura (P_i ? x , P_k ! e) que «matchean» sintácticamente, considerando la totalidad de los procesos de un programa P de CSP, debe cumplirse:

$$(\{p_k\}\ P_i\ ?\ x\ \{q_k\} \wedge \{p_i\}\ P_k\ !\ e\ \{q_i\}) \rightarrow (\{p_k \wedge p_i\}\ P_i\ ?\ x\ ||\ P_k\ !\ e\ \{q_k \wedge\ q_i\}).$$

Es por medio de esta (meta-)regla que se consisten las asunciones de la primera etapa. Su aplicación involucra la utilización del axioma de comunicación. Cuando se cumple el test de cooperación, se dice que las proof outlines secuenciales de la primera etapa *cooperan*.

□

(7) Regla de composición distribuida (DIST):

$$\frac{\{p_i\}\ S^*_i\ \{q_i\},\ 1 \le i \le n,\ \text{son proof outlines secuenciales que cooperan}}{\{\wedge_{i=1,n}\ p_i\}\ [\ ||_{\ i=1,n}\ S_i\ ]\ \{\wedge_{i=1,n}\ q_i\}}$$

Como se observa, la forma de esta (meta-)regla es similar a la de las reglas O-CONC y R-CONC que se vieron para la verificación de los programas concurrentes con variables compartidas. En el caso distribuido, las pruebas locales deben consistirse mediante el test de cooperación.

□

Se cumple que el método AFR es sensato. La prueba se basa en un lema denominado Lema de Preservación Distribuida, similar a los lemas de preservación referidos anteriormente. También se cumple que AFR es relativamente completo. En este caso no va a alcanzar con la regla AUX de variables auxiliares, lo que se verá enseguida.

Se presenta a continuación un primer ejemplo de aplicación del método AFR:

Ejemplo. Instanciando el programa anterior P_{prop} con n = 3, queda:

P_{prop} :: $[P_1 \,||\, P_2 \,||\, P_3]$, siendo:

P_1 :: P_2 ! y_1
P_2 :: P_1 ? y_2 ; P_3 ! y_2
P_3 :: P_2 ? y_3

Se comprueba fácilmente que las siguientes son proof outlines secuenciales que cooperan:

P_1 :: $\{y_1 = Y\}$ P_2 ! y_1 $\{y_1 = Y\}$
P_2 :: $\{true\}$ P_1 ? y_2 $\{y_2 = Y\}$; P_3 ! y_2 $\{y_2 = Y\}$
P_3 :: $\{true\}$ P_2 ? y_3 $\{y_3 = Y\}$

Finalmente, aplicando las reglas DIST y CONS, se prueba en AFR:
$\{y_1 = Y\}$ P_{prop} $\{y_3 = Y\}$.

En el siguiente ejemplo se presenta otra prueba sencilla de correctitud parcial, aplicando el método de verificación AFR.
De todos modos, surge el problema de incompletitud del método. Por lo tanto, se justificará la introducción de una nueva regla para resolverlo:

Ejemplo. Sea el siguiente programa distribuido P de CSP:

P :: [P_1 || P_2], con:

P_1 :: x := 0 ; P_2 ! x ; x := x + 1 ; P_2 ! x
P_2 :: P_1 ? y ; P_1 ? y

Se cumple que $\models$ {true} P {x = 1 $\wedge$ y = 1}, pero sin embargo se puede demostrar que no existe una prueba de esta fórmula de correctitud en AFR. Las proof outlines naturales serían:

P_1 :: {true} x := 0 {x = 0} ; P_2 ! x {x = 0} ; x := x + 1 {x = 1} ; P_2 ! x {x = 1}

P_2 :: {true} P_1 ? y {y = 0} ; P_1 ? y {y = 1}

Aplicando el test de cooperación, debe cumplirse:

(a) {x = 0 $\wedge$ true} P_2 ! x || P_1 ? y {x = 0 $\wedge$ y = 0}
(b) {x = 0 $\wedge$ y = 0} P_2 ! x || P_1 ? y {x = 0 $\wedge$ y = 1}
(c) {x = 1 $\wedge$ true} P_2 ! x || P_1 ? y {x = 1 $\wedge$ y = 0}
(d) {x = 1 $\wedge$ y = 0} P_2 ! x || P_1 ? y {x = 1 $\wedge$ y = 1}

Por COM y CONS valen (a) y (d), pero en cambio los casos (b) y (c) no pasan el test.

El problema radica en que el test de cooperación exige que se consideren todos los pares de instrucciones de lectura y escritura que «matchean» sintácticamente, y desde el punto de vista semántico, los casos (b) y (c) no se producen a lo largo de la ejecución del programa P.
Para resolver la incompletitud, el método AFR plantea reforzar las aserciones con información adicional, de modo tal de evitar chequeos innecesarios. La idea es incorporar en la prueba invariantes globales y variables auxiliares.
Sean, por ejemplo, las variables auxiliares c_1 de P_1 y c_2 de P_2 , inicializadas en 0, tales que en cada momento tengan como valor el número de comunicaciones llevadas a cabo por P_1 y P_2, respectivamente.

Y sea el invariante global IG = ($c_1 = c_2$). Obviamente IG se cumple antes y después de cada comunicación entre P_1 y P_2. Si se amplía P con las variables auxiliares definidas, y se le asignan a éstas valores acordes con la semántica especificada, se obtiene:

P' :: [P'_1 || P'_2], con:

P'_1 :: $\{c_1 = 0\}$ x := 0 ; $\{x = 0 \wedge c_1 = 0\}$
<P_2 ! x ; $c_1 := c_1 + 1$> ; $\{x = 0 \wedge c_1 = 1\}$
x := x + 1 ; $\{x = 1 \wedge c_1 = 1\}$
<P_2 ! x ; $c_1 := c_1 + 1$> $\{x = 1 \wedge c_1 = 2\}$

P'_2 :: $\{c_2 = 0\}$
<P_1 ? y ; $c_2 := c_2 + 1$> ; $\{y = 0 \wedge c_2 = 1\}$
<P_1 ? y ; $c_2 := c_2 + 1$> $\{y = 1 \wedge c_2 = 2\}$

asumiendo que las secuencias internas de las secciones delimitadas por «< >» se ejecutan atómicamente. Planteando ahora el test de cooperación con respecto a estas proof outlines secuenciales y el invariante global IG antes y después de cada comunicación, se debe chequear:

(a) $\{x = 0 \wedge c_1 = 0 \wedge c_2 = 0 \;\; c_1 = c_2\}$
<P_2 ! x ; $c_1 := c_1 + 1$> || <P_1 ? y ; $c_2 := c_2 + 1$>
$\{x = 0 \wedge c_1 = 1 \wedge y = 0 \wedge \;\; c_2 = 1 \wedge c_1 = c_2\}$

(b) $\{x = 0 \wedge c_1 = 0 \wedge y = 0 \wedge c_2 = 1 \wedge c_1 = c_2\}$
<P_2 ! x ; $c_1 := c_1 + 1$> || <P_1 ? y ; $c_2 := c_2 + 1$>
$\{x = 0 \wedge c_1 = 1 \wedge y = 1 \wedge \;\; c_2 = 2 \wedge c_1 = c_2\}$

(c) $\{x = 1 \wedge c_1 = 1 \wedge c_2 = 0 \wedge c_1 = c_2\}$
<P_2 ! x ; $c_1 := c_1 + 1$> || <P_1 ? y ; $c_2 := c_2 + 1$>
$\{x = 1 \wedge c_1 = 2 \wedge y = 0 \wedge \;\; c_2 = 1 \wedge c_1 = c_2\}$

(d) $\{x = 1 \wedge c_1 = 1 \wedge y = 0 \wedge c_2 = 1 \wedge c_1 = c_2\}$
<P_2 ! x ; $c_1 := c_1 + 1$> || <P_1 ? y ; $c_2 := c_2 + 1$>
$\{x = 1 \wedge c_1 = 2 \wedge y = 1 \wedge \;\; c_2 = 2 \wedge c_1 = c_2\}$

Ahora (b) y (c) se cumplen trivialmente, dado que las precondiciones son falsas. La información agregada a la prueba original, que establece que en la ejecución del programa P no pueden producirse determinadas

comunicaciones entre los procesos P_1 y P_2 , ha sido provista por el invariante global IG.

Por otro lado, los casos (a) y (d) se cumplen, considerando una variante de la regla COOP que es la siguiente:

$$\frac{\{p\}\ S_1\ ;\ S_3\ \{p_1\}\ ,\ \{p_1\}\ [\alpha\ ||\ \beta]\ \{p_2\}\ ,\ \{p_2\}\ S_2\ ;\ S_4\ \{q\}}{\{p\}\ [<S_1\ ;\ \alpha\ ;\ S_2> \ ||\ <S_3\ ;\ \beta\ ;\ S_4>]\ \{q\}}$$

tal que:

- α y β son instrucciones de comunicación que «matchean» sintácticamente. La regla debe aplicarse, como antes, considerando todos los pares (α, β) de este tipo, en todos los procesos.
- S_1, S_2, S_3 y S_4 no incluyen instrucciones de comunicación.

Finalmente, para la prueba del programa completo se necesitará también una variante de la regla DIST, que es la siguiente:

$$\frac{\{p_i\}\ S^*_i\ \{q_i\},\ 1 \le i \le n,\ \text{son proof outlines secuenciales que cooperan y utilizan IG}}{\{\wedge_{i=1,n}\ p_i \wedge\ IG\}\ [\ ||_{\ i=1,n}\ S_i\ ]\ \{\wedge_{i=1,n}\ q_i\ \wedge\ IG\}}$$

tal que ninguna variable del invariante global IG es modificada fuera de las secciones delimitadas por «< >».

Aplicando en el ejemplo esta variante de DIST se obtiene:

$\{c_1 = 0 \wedge c_2 = 0 \wedge c_1 = c_2\}\ P'\ \{x = 1 \wedge c_1 = 2 \wedge y = 1 \wedge c_2 = 2 \wedge c_1 = c_2\}$,

y entonces, aplicando las reglas CONS y AUX se llega a lo que se pretendía, es decir:

$$\vdash \{true\}\ P\ \{x = 1 \wedge y = 1\}.$$

Para completar la descripción del método AFR, se deben introducir otras variantes de la regla COOP. Además debería definirse formalmente la semántica de la secciones atómicas delimitadas por «< >» utilizadas en el ejemplo.

Estos temas no se van a desarrollar. El propósito en el ejemplo ha sido introducir el problema de la incompletitud provocada por el test de cooperación, y mencionar la manera de solucionarlo mediante el uso de invariantes globales y variables auxiliares.

3.4.2.3. Verificación de la correctitud total de los programas de CSP y otras consideraciones

La verificación de la correctitud total de los programas de CSP se realiza, como siempre, desarrollando una prueba separada para cada propiedad. En toda prueba se asume que las otras propiedades se cumplen, y se parte de un conjunto de pruebas locales que cooperan, es decir, de una proof outline distribuida.
Para probar la ausencia de deadlock se utiliza la (meta-)regla AFR-DEADLOCK, la cual establece, dado un programa $P :: [||_{i = 1,n} P_i]$:

a) Elaborar una proof outline distribuida de {p} P {q}.
b) Identificar sintácticamente todos los casos posibles C_m de deadlock, es decir, todas las n-tuplas asociadas a los n procesos P_i, compuestas por instrucciones de comunicación α_i y continuaciones sintácticas vacías E_i, de manera tal que al menos haya una instrucción de comunicación, y no exista ningún par (α_i , α_k) de instrucciones que «matcheen» sintácticamente (porque de lo contrario no habría posibilidad de deadlock).
c) Finalmente, definir las imágenes semánticas δ_m de los casos C_m, utilizando las aserciones de la proof outline distribuida, y probar que son falsas.

Ahora, las expresiones δ_m estarán formadas por tres tipos de operandos:

– $post(P_i)$, en el caso de las continuaciones sintácticas vacías E_i.
– $pre(\alpha_i)$, en el caso de las instrucciones de comunicación α_i.
– $pre(S) \wedge (\bigwedge_{i \varepsilon O} B_i) \wedge (\bigwedge_{k \varepsilon C} \neg B_k)$, en el caso de que S sea una selección condicional o una repetición con instrucciones de comunicación formando parte de las guardias, siendo las B_i expresiones booleanas verdaderas con $\varnothing \neq O \subseteq \Delta$, y las B_k expresiones booleanas falsas con $C = \Delta - O$.

□

Ejemplo. Considerando otra vez el programa P_{prop}:

$$P_{prop} :: [P_1 :: P_2 \,!\, y_1 \,||\, P_2 :: P_1 \,?\, y_2 \,;\, P_3 \,!\, y_2 \,||\, P_3 :: P_2 \,?\, y_3],$$

se va a necesitar utilizar un invariante global y variables auxiliares. Se propone la siguiente proof outline distribuida:

$\{w_1 = 1\}$
$[< P_2 \,!\, y_1 \,;\, w_1 := 2>$
$\{w_1 = 2\}$
$||$
$\{w_2 = 0\}$
$<P_1 \,?\, y_2 \,;\, w_2 := 1> \,;$
$\{w_2 = 1\}$
$<P_3 \,!\, y_2 \,;\, w_2 := 2>$
$\{w_2 = 2\}$
$||$
$\{w_3 = 0\}$
$<P_2 \,?\, y_3 \,;\, w_3 := 2>]$
$\{w_3 = 2\}$

La semántica de las variables w es la siguiente: $w_i = 0$ indica que P_i está preparado para leer, $w_i = 1$ que P_i leyó, y $w_i = 2$ que P_i terminó. El invariante global propuesto es:

$IG = [(w_1 = 1 \wedge w_2 = 0 \wedge w_3 = 0) \vee (w_1 = 2 \wedge w_2 = 1 \wedge w_3 = 0) \vee (w_1 = 2 \wedge w_2 = 2 \wedge w_3 = 2)]$.

Un caso posible de deadlock a plantear sería, por ejemplo, cuando P_1 y P_2 están por ejecutar el comando de output, habiendo ya terminado el proceso P_3. Esta situación se describe mediante la siguiente imagen semántica:

$w_1 = 1 \wedge w_2 = 1 \wedge w_3 = 2,$

que contradice el invariante global IG.

Queda como ejercicio para el lector, listar todas las situaciones posibles de deadlock, y demostrar que las imágenes semánticas asociadas son falsas.

Para probar que las selecciones condicionales no fallan se utiliza la regla AFR-COND*, de características similares a la presentada en el método D* para los programas de GCL: se le agrega a la regla AFR-COND de correctitud parcial una premisa que asegure, dada la precondición p, la existencia de alguna guardia verdadera B_i.

Para verificar la no divergencia de los programas de CSP, en las pruebas locales se emplea la regla AFR-REP*, también de características similares a la presentada en el método D*.
Por cada repetición en un proceso P_i, se define un invariante parametrizado p(n). La particularidad en este caso es que el valor de n puede depender de una comunicación entre P_i con otro proceso P_k, y de esta manera debe asumirse y chequearse en el test de cooperación de la segunda etapa de la prueba.
Formalmente, la regla AFR-REP* establece, dados el invariante p, las expresiones booleanas B_i, las instrucciones de comunicación α_i y los subprogramas S_i, con i variando en el conjunto Δ de direcciones,

INIC: $p \rightarrow \exists n: p(n)$

CONT: $p(n+1) \rightarrow \bigvee_i B_i$

DEC: $<p(n) \wedge n > 0 \wedge B_i> \alpha_i ; S_i <\exists k: k < n \wedge p(k)>$

TERM: $p(0) \rightarrow \bigwedge_i \neg B_i$

$<p> *[\Box_i B_i ; \alpha_i \rightarrow S_i] <p(0)>$

□

Por ejemplo, se cumple:

$<0 \leq x \leq 10 \wedge y \neq 0>$
$P :: [P_1 :: *[x \geq 0 ; P_2 ! x \rightarrow x := x - 1]$
$\|$
$P_2 :: *[y \neq 0 ; P_1 ? y \rightarrow skip]]$
$<true>$

Notar en el ejemplo que considerado aisladamente, no se puede asegurar que el proceso P_2 termina. Se debe hacer una asunción que se va a consistir en el test de cooperación de la segunda etapa de la prueba. (Notar además que el programa P puede tener deadlock, si al comienzo el contenido de la variable y es cero.)

Asumiendo la existencia de fairness, la prueba de no divergencia podría basarse en la idea de las direcciones útiles desarrollada antes para los programas de GCL. Pero ahora se considerarían pares de direcciones, relacionadas con las instrucciones de lectura y escritura de las guardias ejecutadas sincrónicamente.

Aún con lenguajes de programación distribuida más complejos, la verificación de los programas puede resolverse con las mismas ideas generales desarrolladas previamente.
Es el caso, por ejemplo, de los lenguajes que permiten establecer comunicaciones entre más de dos procesos (*multiparty interactions*).

Otro ejemplo es el de los programas del lenguaje ACSP, una variante de CSP en que los pasajes de mensajes son asincrónicos. En este caso, la escritura no es causa de deadlock, porque el proceso correspondiente envía el valor a un canal que lo comunica exclusivamente con otro proceso y luego progresa. La lectura en cambio sigue siendo causa de bloqueo, porque el canal solicitado puede estar vacío.
Sintácticamente, para diferenciar el lenguaje ACSP de CSP, la forma de sus instrucciones de comunicación son c_i ?? x para la lectura desde un canal c_i, y c_i !! e para la escritura al canal c_i.
Tal como se hizo en el caso del lenguaje RVL para registrar la situación de los recursos, la semántica de ACSP debe considerar, además de los estados, un arreglo con información de los canales (es decir, los datos enviados por lo distintos procesos). Hay un ejercicio al final de esta parte del libro referido a la semántica de ACSP.
Por ejemplo, sea el siguiente programa P :: [|| P_i] de ACSP, que calcula el máximo a partir de un anillo de procesos. Cada proceso tiene la forma:

P_i :: z_i := 0 ; y_i := 0 ; c_i !! x_i ;
*[$y_i \neq N \rightarrow c_{i-1}$?? y_i ;
[$x_i = y_i \rightarrow z_i := 1$; $y_i := N$; c_i !! N
□
$x_i > y_i \rightarrow$ skip
□
$x_i < y_i \rightarrow c_i$!! y_i]]

El proceso P_i lee del canal c_{i-1} y escribe en el canal c_i (c_n es en realidad el canal c_0, con i variando entre 1 y n). Todos los x_i son distintos, y $N > 0$ es mayor que todos ellos. Sólo un proceso P_i termina con $z_i = 1$, y es el

que inicialmente tenía el mayor x_i. Cada proceso P_i envía al comienzo el valor de su x_i y propaga todo y_i recibido que sea mayor que x_i. De esta manera, solamente el máximo x_i recorre todo el anillo. Después que éste vuelve al proceso de origen P_i, P_i envía un mensaje con el valor N para terminar el programa.

Las pruebas de los programas ACSP también se desarrollan en dos etapas. Como característica diferencial con respecto a lo visto para CSP, se asocia a cada canal c un par de variables IN_c y OUT_c, cuya semántica se refiere a la cantidad y el orden de los mensajes recibidos y enviados. Con la utilización de estas variables se simplifica el test de cooperación, dado que no sería necesario en general introducir invariantes globales: mediante una premisa que relacione las variables IN_c y OUT_c, los casos de «matching» sintáctico que no se producen semánticamente podrían probarse trivialmente.

Apéndice 3

A3.1. Verificación de programas con procedimientos

En este capítulo se trata suscintamente la verificación de programas con procedimientos.
Para simplificar la presentación, se considera solamente una extensión del lenguaje PLW, y por lo tanto la verificación de programas se entenderá en el marco de los métodos H y H* ampliados con reglas relacionadas con procedimientos.
Para introducir gradualmente el tema, primero se consideran procedimientos sin parámetros ni recursión (Sección A3.1.1), y luego se incorporan dichas características (Secciones A3.1.2 y A3.1.3).

A3.1.1. Procedimientos sin parámetros ni recursión

Se va a trabajar con el lenguaje de programación PROC, que es el lenguaje PLW ampliado con procedimientos. Un programa P de PROC tiene la siguiente forma:

P :: D ; S
D :: procedure <proc-1> : S_1 end , … , procedure <proc-n> : S_n end
S :: call <proc> | skip | x := e | S_1 ; S_2 | ...

D es una lista de declaraciones de procedimientos. Cada S_i, con $1 \leq i \leq n$, es el cuerpo del procedimiento de nombre proc-i.
S es un subprograma con instrucciones de PLW más la instrucción call, mediante la cual puede invocar a procedimientos declarados en la lista D. Por ahora, para simplificar la introducción del tema, no se va a permitir que los procedimientos se invoquen entre sí. Es decir, inicialmente se cumple que para todo i, con 1 $i \leq n$, $S_i \varepsilon$ PLW.
Informalmente, cuando S invoca al procedimiento proc-i, se ejecuta su cuerpo S_i, y a su término se continúa con la ejecución desde la instrucción siguiente al call. Las variables de S y de los procedimientos pertenecen a un único estado global.

Ejemplo. El programa de PROC:

P1 :: procedure suma1 : x := x + 1 end ;
while y > 0 do call suma1 ; y := y – 1 od

suma en x su valor inicial al valor de y, que no debe ser negativo.

La semántica operacional de la instrucción call se define mediante la relación «$\rightarrow$» de la siguiente manera:

$$\langle \text{call proc}, \sigma \rangle \rightarrow \langle S, \sigma \rangle,$$

tal que el procedimiento proc está declarado en el programa de referencia, y S es el cuerpo de proc. Por lo tanto, se cumple que:

$$\text{val}(\pi(\text{call proc}, \sigma)) = \text{val}(\pi(S, \sigma)).$$

En realidad, las configuraciones tienen tres componentes: además de la continuación sintáctica y el estado corriente, debe considerarse el entorno (*environment*), que contiene las declaraciones de procedimientos. Dado que las transiciones no modifican el entorno, entonces para simplificar la notación se lo puede omitir en las configuraciones.

□

Si Ide_Proc es el conjunto de los nombres de procedimientos, y Ent (por entorno) es el conjunto de las declaraciones de procedimientos, es decir:

$$\text{Ent} = (\text{Ide\_Proc} \rightarrow (\Sigma \rightarrow \Sigma)),$$

entonces los dominios que integran la función semántica $\mathcal{M}$ asociada a PROC son los siguientes:

$$\mathcal{M} : \text{PROC} \rightarrow (\text{Ent} \rightarrow (\Sigma \rightarrow \Sigma)).$$

Con las consideraciones anteriores, se presenta a continuación la regla de verificación de correctitud parcial asociada a la instrucción call (INVOC), que amplía el método H:

$$\frac{\{p\}\ S\ \{q\}}{\{p\}\ \text{call proc}\ \{q\}}$$

La regla establece que si se cumple q cuando la ejecución de S termina a partir de p, siendo S el cuerpo del procedimiento proc, entonces también se cumple q cuando la invocación al procedimiento proc termina a partir de p.

□

Ejercicio. *Probar que H ∪ {INVOC} sigue siendo sensato y relativamente completo.*

□

Como el while es la única instrucción que puede causar no terminación, no se necesita definir ninguna regla adicional para la verificación de la correctitud total de los programas de PROC.

Ejemplo. Volviendo al programa P1 anterior, se prueba en H ∪ {INVOC} que:

$\{x = X \wedge y = Y \wedge Y \geq 0\}$
P1 :: procedure suma1 : x := x + 1 end ;
while $y > 0$ do call suma1 ; y := y – 1 od
$\{x = X + Y\}$

Algunos pasos representativos de la prueba son los siguientes:

(1) $\{x + 1 + y – 1 = X + Y \wedge y – 1 \geq 0\}$ x := x + 1 $\{x + y – 1 = X + Y \wedge y – 1 \geq 0\}$
(ASI)
(2) $\{x + 1 + y – 1 = X + Y \wedge y – 1 \geq 0\}$ call suma1 $\{x + y – 1 = X + Y \wedge y – 1 \geq 0\}$
(INVOC, 1)
- -
(3) $\{x + y = X + Y \wedge y \geq 0\}$ while $y > 0$ do call suma1 ; y := y – 1 od $\{x = X + Y\}$
(SEC, 2, REP, etc.)
(4) $\{x = X \wedge y = Y \wedge Y \geq 0\}$ P1 $\{x = X + Y\}$
(CONS, 3)

Ejercicio. *Completar la prueba del ejemplo.*

□

A3.1.2. Procedimientos con parámetros y sin recursión

En esta sección se amplía el lenguaje PROC con la posibilidad de parametrizar los procedimientos. Se mantiene la idea de la prueba establecida por la regla INVOC, pero ahora la relación entre las fórmulas de correctitud asociadas al cuerpo de un procedimiento y a su invocación, se define a través de sustituciones de variables y expresiones.
La nueva forma de un programa P de PROC es:

P :: D ; S
D :: procedure <proc-1> (val $\mathbb{Y}_1$; var $\mathbb{X}_1$) : S_1 end ,
- -
procedure <proc-n> (val $\mathbb{Y}_n$; var $\mathbb{X}_n$) : S_n end
S :: call <proc> ($\mathbb{E}$, $\mathbb{V}$) | skip | x := e | S_1 ; S_2 | ...

Los procedimientos se declaran con dos listas de parámetros formales, y las invocaciones incluyen dos listas de parámetros reales. Los términos *val* y *var* se refieren, respectivamente, al pasaje de parámetros por *valor* y por *valor-resultado*. La cantidad de parámetros reales y formales debe coincidir en los dos casos. Los parámetros formales son todos distintos.
Dada la declaración del procedimiento proc, la lista $\mathbb{Y}$ enumera sus parámetros formales $y_1, y_2, \ldots, y_m$, asociados al pasaje por valor. Estos parámetros reciben valores enteros de los parámetros reales enumerados en la lista $\mathbb{E}$ de expresiones enteras $e_1, e_2, \ldots, e_m$.
Por su parte, la lista $\mathbb{X}$ del procedimiento proc enumera sus parámetros formales $x_1, x_2, \ldots, x_k$, asociados al pasaje por valor-resultado, los cuales reciben valores enteros de los parámetros reales enumerados en la lista $\mathbb{V}$ de variables enteras $v_1, v_2, \ldots, v_k$.
Los parámetros y_i reciben valores iniciales al momento de la invocación (o activación) del procedimiento. Los parámetros x_i también reciben valores iniciales, y cuando el control retorna al invocante, sus valores finales se devuelven a los parámetros reales v_i.
Los parámetros formales son locales al procedimiento, y existen solamente en el período que va de la activación a la desactivación (o retorno) del procedimiento. De esta manera, el estado corriente se amplía al momento de la invocación, y se reduce cuando el control retorna al invocante.
Ahora se permiten invocaciones entre procedimientos, pero todavía sin que se produzcan encadenamientos cíclicos de llamadas que provoquen recursión, para mantener simplificada la presentación.

□

Ejemplo. El programa de PROC:

```
P2 :: procedure incdec (var i, x) :
        while i > 0 do x := x + 1 ; i := i – 1 od end ;
        call incdec (a, b)
```

suma en b su valor inicial al valor de a, que no debe ser negativo. Se utiliza el pasaje de parámetros por valor-resultado. Notar que P2 se comporta como el programa P1 de la sección anterior, pero ahora se utiliza parametrización.

Con las consideraciones anteriores, la siguiente sería una posible regla de verificación de correctitud parcial para el call a un procedimiento con parámetros:

$$\frac{p \rightarrow p'^{\,Y\ X}_{\ E\ V}\ ,\ \{p'\}\ S\ \{q'\}\ ,\ q'^{\,X}_{\ V} \rightarrow q}{\{p\}\ \text{call proc}\ (E,V)\ \{q\}}$$

tal que:

- Las aserciones p y q se expresan en términos de las variables de E y V.
- La aserción p' se expresa en términos de las variables de Y y X.
- La aserción q' se expresa en términos de las variables de X.
- La expresión $p'^{\,Y\ X}_{\ E\ V}$ denota la sustitución en p' de las variables libres de Y y X por las expresiones de E y las variables de V, respectivamente.
- La expresión $q'^{\,X}_{\ V}$ denota la sustitución en q' de las variables libres de X por las variables de V.
- El cuerpo S sólo utiliza los parámetros de Y y X, es decir que no hace referencia a variables locales ni globales.

La regla se denomina PAR, y pareciera adecuada de acuerdo a la semántica descripta. La aserción q' se expresa sólo en términos de los parámetros por valor-resultado, porque son los únicos desde los que se devuelven valores al invocante. La restricción de que el cuerpo S sólo utilice parámetros, es también a los efectos de simplificar la presentación.
Sin embargo, la regla PAR no es sensata, como lo muestra el siguiente ejemplo. Sea el siguiente programa P para incrementar en uno una variable:

P :: procedure inc1 (var x_1, x_2) : x_1 := x_2 + 1 end ;
call inc1 (a, a)

Utilizando la regla PAR, se prueba en H $\cup$ {PAR} la fórmula:

{true} call inc1 (a, a) {a = a + 1},

que obviamente no es verdadera. Dicha fórmula se obtiene asumiendo:

- p = true,
- q = (a = a + 1),
- p' = true,
- q' = (x_1 = x_2 + 1),

porque se hace:

(1) {true} x_1 := x_2 + 1 {x_1 = x_2 + 1}	(ASI, CONS)
(2) true $\rightarrow$ true $^{x1,\ x2}$	(MAT)
(3) (x_1 = x_2 + 1) $^{x1,\ x2}_{a,\ a}$ $\rightarrow$ (a = a + 1)	(MAT)
(4) {true} call inc1 (a, a) {a = a + 1}	(PAR, 1, 2, 3)

El problema se produce porque los parámetros reales son iguales (lo que se conoce como *alias*), y sus valores se asignan a parámetros formales que se suponen distintos.

Por lo tanto, una manera de corregir la regla PAR anterior para que sea sensata, será agregarle como premisa la condición:

disjoint ($\mathbb{V}$),

que establece que los componentes de la lista $\mathbb{V}$ son distintos.

□

Existe otra manera, que no se va a describir, que evita restringir la invocación con alias a procedimientos. La idea consiste en aplicar la regla PAR recién después de transformar sintácticamente, de un modo adecuado, la invocación y la declaración del procedimiento correspondiente. Hay un ejercicio sobre dicha idea al final de esta parte del libro.

Las restricciones mencionadas limitan las posibilidades del uso de procedimientos con parámetros. Se han establecido para introducir fácilmente la idea básica de verificación en este marco. De todos modos, cabe remarcar que la flexibilidad provista a través de distintos tipos de pasajes de parámetros (por referencia, por nombre, etc.), en combinación con la recursión y el uso de variables locales y globales, atenta contra la sensatez y completitud relativa de los métodos de verificación asociados.
El siguiente es un ejemplo de aplicación de la regla PAR, que cierra la sección. Al final del capítulo se presenta una prueba que trata conjuntamente la parametrización y la recursión.

Ejemplo. Dado el programa P2 anterior, se va a probar en $H \cup \{PAR\}$:

$\{a = N \geq 0 \wedge b = 0\}$
P2 :: procedure incdec (var i, x) :
while $i > 0$ do $x := x + 1$; $i := i - 1$ od end ;
call incdec (a, b)
$\{a = 0 \wedge b = N\}$

Haciendo:

- $p = (a = N \geq 0 \wedge b = 0)$,
- $q = (a = 0 \wedge b = N)$,
- $p' = (i = N \geq 0 \wedge x = 0)$,
- $q' = (i = 0 \wedge x = N)$,

algunos pasos representativos de la prueba son:

(1) $\{i = N \geq 0 \wedge x = 0\}$ while $i > 0$ do $x := x + 1$; $i := i - 1$ od $\{i = 0 \wedge x = N\}$	(REP, CONS, etc.)
(2) $(a = N \geq 0 \wedge b = 0) \rightarrow (i = N \geq 0 \wedge x = 0)^{i, x}_{a, b}$	(MAT)
(3) $(i = 0 \wedge x = N)^{i, x}_{a, b} \rightarrow (a = 0 \wedge b = N)$	(MAT)
(4) disjoint (a, b)	
(5) $\{a = N \geq 0 \wedge b = 0\}$ P2 $\{a = 0 \wedge b = N\}$	(PAR, 1, 2, 3, 4)

Ejercicio. *Completar la prueba del ejemplo.*

□

A3.1.3. Procedimientos con recursión

Finalmente, en esta sección se trata la verificación de programas con procedimientos recursivos. Primero se plantea una regla para el caso en que no hay parametrización, y luego se presenta un ejemplo en que se combinan las dos posibilidades.
Sigue valiendo la semántica del lenguaje PROC descripta anteriormente. Para simplificar la presentación se va a considerar sólo la recursión simple, en que un procedimiento se puede invocar solamente a sí mismo (es decir, no hay recursión por invocación mutua entre procedimientos).
La regla INVOC planteada anteriormente para la correctitud parcial no sirve en este caso. Según dicha regla, para probar la fórmula {p} call proc {q} se debe probar {p} S {q}, siendo S el cuerpo de proc. Pero si S incluye la instrucción call proc, entonces en algún momento se deberá probar nuevamente {p} call proc {q}, y luego otra vez {p} S {q}, y así sucesivamente sin terminar nunca.
La solución propuesta es definir otra regla, denominada REC, tal que su propia forma represente la ejecución recursiva de procedimientos. La idea es simple:

- Dado el procedimiento proc con cuerpo S que incluye la instrucción call proc,
- si asumiendo que se cumple {p} call proc {q} se prueba {p} S {q}, entendiendo al call proc como interno de S,
- entonces quiere decir que se cumple {p} call proc {q}, entendiendo ahora al call proc como el invocante de S.

Es decir, la forma de la regla REC es la siguiente:

$$\frac{Tr, \{p\}\ \text{call proc}\ \{q\} \vdash \{p\}\ S\ \{q\}}{\{p\}\ \text{call proc}\ \{q\}}$$

donde el call proc de la premisa es interno de S, y el call proc de la conclusión es el invocante de S.

□

Se cumple que $H \cup \{REC\}$ es sensato y relativamente completo. Una manera inductiva de probar la sensatez de la regla REC se basa en la profundidad h de la recursión (h es la cantidad de invocaciones). Informalmente, la idea general sería:

Si h = 1, entonces el cuerpo S no invoca a proc. De esta manera, si se prueba {p} S {q}, quiere decir por la semántica del call que se cumple {p} call proc {q}, siendo el call proc el invocante de S.

Si h = k, y si asumiendo que se cumple {p} call proc {q} se llega a probar {p} S {q}, siendo el call proc interno de S, entonces quiere decir por la semántica del call que vale {p} call proc {q}, siendo el call proc el invocante de S. Es decir que asumiendo {p} call proc {q}, con h = k, se prueba la fórmula {p} call proc {q}, con h = k + 1.

Ejemplo. Se prueba en $H \cup \{REC\}$:

$\{x \geq 0\}$
P_{fac1} :: procedure fac1 :
if x = 0 then y := 1
else x := x – 1 ; call fac1 ; x := x + 1 ; y := y **.** x fi end ;
call fac1
$\{y = x!\}$

es decir, el programa P_{fac1} obtiene en y el factorial de un número natural x, valiéndose del procedimiento recursivo fac1.

Algunos pasos representativos de la prueba son:

(1) $\{x \geq 0 \wedge x = 0\}$ y := 1 $\{y = x!\}$ (ASI, CONS)
- -
(2) $\{x \geq 0\}$ call fac1 $\{y = x!\}$ (Asunción)
- -
(3) $\{x \geq 0 \wedge x \neq 0\}$ x := x – 1 ; call fac1 ; x := x + 1 ; y := y **.** x $\{y = x!\}$
(SEC, 2, CONS, etc.)

Si S es el cuerpo del procedimiento fac1, queda:

(4) $\{x \geq 0\}$ S $\{y = x!\}$ (COND, 1, 3)
(5) $\{x \geq 0\}$ P_{fac1} $\{y = x!\}$ (REC, 1 - 4)

Ejercicio. *Completar la prueba del ejemplo.*

□

Ahora, un programa de PROC puede diverger no sólo por los while sino también por los procedimientos recursivos. Se tiene que agregar, por lo tanto, una regla REC* asociada a la terminación de las invocaciones a procedimientos.
La regla que se utiliza es la siguiente:

$$\frac{\forall n: [Tr, \langle p(n)\rangle \text{ call proc } \langle q\rangle \vdash \langle p(n+1)\rangle\ S\ \langle q\rangle] \ ,\ \neg\, p(0)}{\langle \exists n: p(n)\rangle \text{ call proc } \langle q\rangle}$$

Como se observa, REC* se basa en la forma de la regla REC para la verificación de la correctitud parcial. El variante n no aparece libre en las aserciones p ni q, ni es una variable del cuerpo S de proc. Se lo cuantifica universalmente en la primera premisa porque es la misma variable en las dos fórmulas de correctitud. En lo referente a la terminación, REC* establece lo siguiente:

- Si asumiendo que la ejecución del call proc interno del cuerpo S termina después de n invocaciones, se prueba que la ejecución del call proc invocante de S termina luego de n + 1 invocaciones,
- y si cuando n = 0 no hay invocación a S,
- entonces se cumple que la ejecución inicial de la instrucción call proc termina (después de una determinada cantidad de invocaciones).

La premisa $\neg\, p(0)$ establece que cuando n = 0, no hay más invocaciones al cuerpo S. Su función es la misma que la de la implicación $p(0) \rightarrow \neg\, B$ utilizada en la prueba de terminación del while.

□

El método $H^* \cup \{REC^*\}$ es sensato y relativamente completo. La prueba de sensatez de la regla REC* es similar a la de la regla REP*: suponiendo que hay un encadenamiento infinito de llamadas al procedimiento proc, se llega al absurdo de encontrar una cadena descendente infinita en (N, <).

Ejemplo. Se prueba en $H^* \cup \{REC^*\}$ la terminación del programa P_{fac1} anterior, es decir:

<x ≥ 0>
P_{fac1} :: procedure fac1 :
if x = 0 then y := 1
else x := x – 1 ; call fac1 ; x := x + 1 ; y := y . x fi end ;
call fac1
<true>

Se propone como invariante parametrizado: p(n) = (x ≥ 0 ∧ n = x + 1).

Se cumple ¬ p(0), porque (x ≥ 0 ∧ 0 = x + 1) = false.

Además, asumiendo <p(n)> call fac1 <true>, se puede probar <p(n + 1)> S <true>, siendo S el cuerpo de fac1. Algunos pasos representativos de la prueba son los siguientes:

(1) <x ≥ 0 ∧ n + 1 = x + 1 ∧ x = 0> y := 1 <true> (ASI*, CONS*)
- -
(2) <x ≥ 0 ∧ n = x + 1> call fac1 <true> (Asunción)
- -
(3) <x ≥ 0 ∧ n + 1 = x + 1 ∧ x ≠ 0> x := x – 1 ; call fac1 ; x := x + 1 ; y := y . x <true>
(SEC*, 2, CONS*, etc.)

Finalmente:

(4) <x ≥ 0 ∧ n + 1 = x + 1> S <true> (COND*, 1, 3)
(5) <x ≥ 0> P_{fac1} <true> (REC*, 1 - 4, CONS*)

Para cerrar este capítulo, se presenta un ejemplo de prueba de correctitud parcial de un programa que utiliza un procedimiento recursivo con parámetros.
Se va a utilizar la regla PAR-REC, que combina las ideas de las reglas PAR y REC descriptas anteriormente. La regla es la siguiente:

1) $p \rightarrow p'^{\,Y\ X}_{\,E\ V}$
2) $q'^{\,X}_{\,V} \rightarrow q$
3) disjoint (V)
4) Tr, {p'} call proc (Y, X) {q'} |– {p'} S {q'}

{p} call proc (E, V) {q}

tal que:

- Las aserciones p y q se expresan en términos de las variables de $\mathbb{E}$ y $\mathbb{V}$.
- La aserción p' se expresa en términos de las variables de $\mathbb{Y}$ y $\mathbb{X}$.
- La aserción q' se expresa en términos de las variables de $\mathbb{X}$.
- La expresión p' $^{\mathbb{Y}\ \mathbb{X}}_{\mathbb{E}\ \mathbb{V}}$ denota la sustitución en p' de las variables libres de $\mathbb{Y}$ y $\mathbb{X}$ por las expresiones de $\mathbb{E}$ y las variables de $\mathbb{V}$, respectivamente.
- La expresión q' $^{\mathbb{X}}_{\mathbb{V}}$ denota la sustitución en q' de las variables libres de $\mathbb{X}$ por las variables de $\mathbb{V}$.
- La expresión disjoint ($\mathbb{V}$) establece que los componentes de la lista $\mathbb{V}$ son distintos.
- El cuerpo S sólo utiliza los parámetros de $\mathbb{Y}$ y $\mathbb{X}$, es decir que no hace referencia a variables locales ni globales.
- El call proc de la cuarta premisa debe interpretarse como interno del cuerpo S, y el call proc de la conclusión como invocante de S.

□

Ejemplo. Se prueba en H ∪ {PAR-REC}:

```
            {b ≥ 0}
P_fac2 ::   procedure fac2 (val y ; var x) :
            if y = 0 then x := 1
                    else call fac2 (y – 1, x) ; x := x . y fi end ;
            call fac2 (b, a)
            {a = b!}
```

es decir, que el programa P_{fac2} obtiene en la variable a el factorial de un número natural b, valiéndose del procedimiento recursivo fac2 con parámetros.

Se tiene:

- p = b ≥ 0,
- q = (a = b!).

Como la postcondición q' del cuerpo S de fac2 no puede expresarse en términos del parámetro y, entonces para que pueda cumplirse la segunda

premisa de la regla PAR-REC, es decir la implicación $q'^{x}_{a} \rightarrow q$, se debe eliminar el parámetro b de la postcondición q de P_{fac2}. Para tal fin, se va a utilizar la variable de especificación N, quedando:

– p = (b = N ∧ N ≥ 0),
– q = (a = N!).

Se debe probar, según lo que establece la regla PAR-REC:

1) $b = N \wedge N \geq 0 \rightarrow (y = N \wedge N \geq 0)^{y\ x}_{b\ a}$
2) $(x = N!)^{x}_{a} \rightarrow a = N!$
3) disjoint (a)
4) {y = N ∧ N ≥ 0} S {x = N!}, a partir de una asunción adecuada.

La verificación de (1), (2) y (3) es trivial. A continuación se presenta la prueba de (4):

(4.1) {y = N ∧ N ≥ 0 ∧ y = 0} x := 1 {x = N! ∧ y = N ∧ N ≥ 0} (ASI, CONS)
(4.2) {x = (N – 1)! ∧ y – 1 = N – 1 ∧ N – 1 ≥ 0} x := x . y {x = N! ∧ y = N ∧ N ≥ 0} (ASI, CONS)
(4.3) {y – 1 = N – 1 ∧ N – 1 ≥ 0} call fac2 (y – 1, x) {x = (N – 1)! ∧ y – 1 = N – 1 ∧ N – 1 ≥ 0} (Asunción)

Antes de continuar, notar que la relación entre la precondición y la postcondición en el paso (4.3) se establece mediante N – 1, mientras que en la premisa (4) se expresa a través de la variable N. Esto es posible, dado que la premisa (4) de la regla PAR-REC:

Tr, {p'} call proc (Y, X) {q'} |– {p'} S {q'} ,

debe interpretarse como:

Tr, ∀N [{p'} call proc (Y, X) {q'}] |– ∀N [{p'} S {q'}] ,

es decir, no existe ninguna relación entre la variable N de una y otra fórmula de correctitud. N es una variable de especificación, que se utiliza (en este caso) para expresar que a partir de la asunción de que la instrucción call fac2 satisface <p', q'>, debe probarse que el cuerpo S satisface el mismo par <p', q'>. En otras palabras, a lo largo de las invocaciones recursivas al procedimiento proc, sólo se requiere que se mantenga la relación entre las aserciones p' y q'.

La prueba continúa de la siguiente manera:

(4.4) $\{y = N \wedge N \geq 0 \wedge y \neq 0\}$ call fac2 $(y - 1, x)$; $x := x \,.\, y$ $\{x = N! \wedge y = N \wedge N \geq 0\}$ (SEC, 4.2, 4.3, CONS)

(4.5) $\{y = N \wedge N \geq 0\}$ S $\{x = N!\}$ (COND, 4.1, 4.4, CONS)

Por lo tanto, como se cumplen las cuatro premisas de la rega PAR-REC, se ha probado:

$\{b = N \wedge N \geq 0\}$ P_{fac2} $\{a = N!\}$.

A3.2. Verificación de programas reactivos utilizando lógica temporal

En este capítulo se presenta una breve introducción a la verificación de los *programas reactivos* utilizando la *lógica temporal*.
En un programa reactivo no existe la noción de terminación. Su ejecución consiste en una interacción continua con el entorno en el que se ejecuta, y su correctitud se define en términos de dicha interacción. Representantes clásicos de este tipo de programas son los sistemas operativos y los sistemas de administración de bases de datos.
Es de aceptación general que la lógica temporal es un marco adecuado para verificar (y también para especificar y desarrollar de manera sistemática) programas reactivos. En realidad, también se la considera adecuada para la verificación de los programas concurrentes en general.

Primero se define, en la Sección A3.2.1, el lenguaje de programación con el que se va a trabajar (en este caso se va a introducir directamente un modelo computacional). Luego se describe la lógica temporal (Sección A3.2.2). En la Sección A3.2.3 se introduce el método de verificación por medio de un ejemplo. Dado el carácter introductorio de este capítulo, se van a describir solamente algunos pocos axiomas y reglas representativos del método. Al final del capítulo, en la Sección A3.2.4, se hace mención a la verificación automática de programas utilizando la lógica temporal.

A3.2.1. Modelo computacional

Se va a trabajar con programas reactivos concurrentes con variables compartidas (podría llevarse a cabo un desarrollo similar utilizando el modelo distribuido), y se va a utilizar una representación gráfica de los programas.
Los programas están compuestos por procesos secuenciales, que se ejecutan concurrentemente compartiendo un conjunto de variables.
Cada proceso secuencial se representa por un grafo orientado denominado *diagrama de transiciones de estados*. Los arcos y nodos de los grafos se denominan *transiciones* y *locaciones,* respectivamente. Las transiciones representan comandos con guardia atómicos, y las locaciones, distintas posiciones del control del proceso.
Un comando con guardia atómico tiene la forma B $\rightarrow$ S, tal que B es una expresión booleana y S es el skip o una asignación x := e (también se permiten asignaciones paralelas). Las variables x, y las expresiones e, son enteras.
La figura A3.2.1 presenta un ejemplo de programa P con dos procesos secuenciales P_1 y P_2:

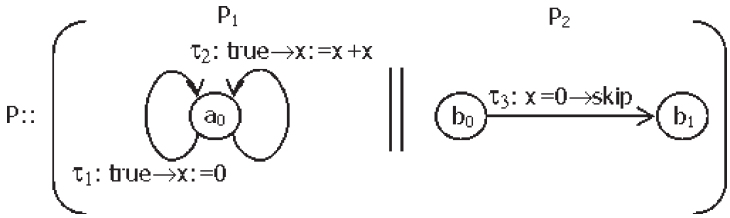


Figura A3.2.1

Informalmente, la semántica se define de la siguiente manera (utilizando el ejemplo de la figura A3.2.1):
Si (el control de) el proceso P_2 está posicionado en la locación b_0, entonces P_2 puede ejecutar atómicamente el comando con guardia x = 0 $\rightarrow$ skip. P_2 sólo puede progresar si se cumple x = 0 (se dirá en este caso que la transición τ_3 está habilitada), y si lo hace, P_2 queda posicionado en la locación b_1.

El proceso P_1 puede progresar de la locación a_0 a la misma locación a_0, por la transición τ_1 o bien por la transición τ_2. La elección de la transición es no determinística.
Al comienzo, los procesos P_1 y P_2 están posicionados, respectivamente, en las locaciones iniciales a_0 y b_0.
Dado un estado inicial global para todos los procesos, habrá una o más computaciones posibles de P, de acuerdo al interleaving de las instrucciones atómicas. Se asume la existencia de fairness (débil o fuerte). Notar en el ejemplo que todas las computaciones posibles de P son infinitas, cualquiera sea el valor inicial de x. Si existe fairness débil, el proceso P_2 termina –sin fairness, P_2 puede no ejecutarse nunca, y también puede suceder que el proceso P_1 nunca progrese por la transición τ_1 (lo mismo aplica a la transición τ_2)–.
El modelo computacional descripto se va a referir de ahora en más como *modelo computacional concurrente*, o MCC. Se presenta a continuación otro ejemplo de programa de MCC:

Ejemplo. Sea el programa concurrente P de la figura A3.2.2:

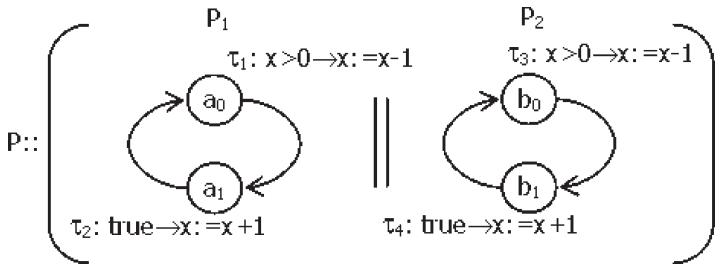


Figura A3.2.2

Se cumple que si al comienzo x = 1, los procesos P_1 y P_2 nunca están simultáneamente posicionados en las locaciones a_1 y b_1, respectivamente (representan las secciones críticas de los procesos), y todas las computaciones posibles del programa P son infinitas. Además, sólo si existe fairness fuerte, los procesos P_1 y P_2 pueden acceder infinitas veces a las locaciones a_1 y b_1, respectivamente.

Dado un programa P :: [|| P_i] de MCC, X_P denota el conjunto de todas las variables, T_P denota el conjunto de todas las transiciones, y Π_P denota el conjunto de todas las posibles computaciones del programa. Cuando queda claro por contexto, puede omitirse el indicador P.

Para especificar la posición de los procesos P_i de P con respecto a las locaciones, se utilizan variables de control c_i. La expresión $c_i = l_{ik}$ significa que P_i está posicionado en la locación l_{ik}. Se define por convención que al comienzo, $c_i = l_{i0}$, para todo i. El conjunto de todas las variables de control de P se denota con C_P (ó directamente C si por contexto queda claro).

La condición inicial θ de un programa P, establece los valores iniciales de las variables de X y de C, y se expresa de la siguiente manera:

$$\theta = \sigma_0 \wedge \text{first} ,$$

tal que $\sigma_0 = \vartheta(x_1, x_2, \ldots, x_m)$ asigna valores iniciales a las variables de programa, y la expresión first denota que todos los procesos están posicionados en sus locaciones iniciales, es decir:

$$\text{first} = (c_1 = l_{10} \wedge c_2 = l_{20} \wedge \ldots \wedge c_n = l_{n0}) .$$

Verificar un programa P con respecto a una especificación Φ será probar, una a una, las propiedades p que componen Φ, considerando $\{\theta\}$ P :: [|| P_i]. En otras palabras, habrá que demostrar para todo p, que p se cumple en todas las computaciones posibles de P, a partir de la condición inicial θ (se verá enseguida que otra posibilidad será demostrar p en un árbol de computaciones). Para simplificar la notación, en lugar de utilizar la expresión completa θ , se puede emplear solamente el estado inicial σ_0.

A3.2.2. Lógica temporal

La lógica temporal extiende la lógica clásica con operadores temporales. Con ella se pueden expresar adecuadamente, utilizando una notación que se explicará enseguida, propiedades como:

– Exclusión mutua. Sea la fórmula f = $\mathbb{G} \neg (SC_1 \wedge SC_2)$. Si $\mathbb{G}$ es el operador temporal «siempre», y SC_i significa que el proceso P_i está en su sección crítica, entonces esta expresión establece que P_1 y P_2 nunca están simultáneamente en sus secciones críticas.

– Ausencia de inanición. Sea la fórmula f = $\mathbb{G}$ ($P_{i\,k} \rightarrow \mathbb{F}\ U_{i\,k}$). Si $\mathbb{F}$ es el operador temporal «alguna vez en el futuro», $P_{i\,k}$ significa que el proceso P_i solicita el recurso k, y $U_{i\,k}$ significa que P_i utiliza el recurso k, entonces esta expresión establece que siempre que un proceso solicita un recurso, alguna vez lo utiliza.

– Fairness débil. Sea la fórmula f = $\mathbb{FG}\ H_i \rightarrow \mathbb{GF}\ E_i$. Si H_i significa que la transición τ_i está habilitada, y E_i que el comando con guardia asociado a τ_i se ejecuta, entonces esta expresión establece que si a partir de un momento dado una transición está siempre habilitada, entonces el comando con guardia asociado se va a ejecutar infinitas veces.

– Fairness fuerte. Sea la fórmula f = $\mathbb{GF}\ H_i \rightarrow \mathbb{GF}\ E_i$. Esta expresión establece que si una transición está habilitada infinitas veces, entonces el comando con guardia asociado se va a ejecutar infinitas veces.

Estas propiedades deben verificarse a lo largo de computaciones, es decir de secuencias de estados de la forma mostrada en la figura A3.2.3:

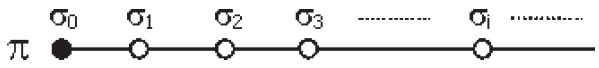


Figura A3.2.3

Si π es finita, se va a indicar $|\pi| = k + 1$, siendo σ_k el estado final de π.
La lógica temporal puede ser *lineal* (*linear temporal logic*) o *ramificada* (*branching temporal logic*), según cómo se interpreten sus fórmulas. Dado un programa P y un estado inicial σ_0, en el caso de la lógica temporal lineal una fórmula se interpreta sobre todas las secuencias de estados con origen en σ_0, y en el caso de la lógica temporal ramificada, la interpretación se lleva a cabo considerando el árbol de estados con raíz en σ_0.
En lo que sigue, se va a trabajar con la lógica temporal lineal. Al final del capítulo se hará alguna mención a la lógica temporal ramificada. El lenguaje L de la lógica temporal lineal que se va a utilizar se define de la siguiente manera. Una fórmula f ε L sii:

a) f es una fórmula de la lógica de primer orden con igualdad (sobre el dominio de los números enteros). Es decir, f ε Assn. En este caso, f se denomina *fórmula de estado*. O bien:

b) f se obtiene a partir de fórmulas de estado utilizando el conectivo $\neg$, el conectivo $\vee$, o los siguientes operadores temporales:

– El operador de futuro $\mathbb{X}$, por *next* (siguiente). Informalmente, su semántica es: dado el estado σ_i de la computación π, se cumple $\mathbb{X}$p en σ_i si se cumple p en el estado siguiente, siendo p una fórmula temporal.

– El operador de pasado $\mathbb{Y}$, por *yesterday* (se lo conoce también como *previous*, es decir anterior). Informalmente, su semántica es: dado el estado σ_i de la computación π, se cumple $\mathbb{Y}$p en σ_i si se cumple p en el estado anterior, siendo p una fórmula temporal. Desde el punto de vista temporal, entonces es el inverso de $\mathbb{X}$.

– El operador de futuro $\mathbb{U}$, por *until* (hasta). Informalmente, su semántica es: dado el estado σ_i de la computación π, se cumple p $\mathbb{U}$ q en σ_i, si se cumple q en un estado futuro σ_k, y se cumple p en los estados σ_i, σ_{i+1}, ... , σ_{k-1}, siendo p y q fórmulas temporales.

– El operador de pasado $\mathbb{S}$, por *since* (desde). Informalmente, su semántica es: dado el estado σ_i de la computación π, se cumple p $\mathbb{S}$ q en σ_i, si se cumple q en un estado pasado σ_k, y se cumple p en los estados σ_{k+1}, σ_{k+2}, ... , σ_i, siendo p y q fórmulas temporales. Desde el punto de vista temporal, entonces es el inverso de $\mathbb{U}$.

□

Una fórmula f de L puede interpretarse a partir de cualquier estado σ_i de una computación π. La expresión:

$$(\pi, i) \models_T f ,$$

establece que la fórmula f se cumple en la computación π a partir de su estado σ_i. Cuando i = 0, se puede escribir directamente $\pi \models_T$ f. Además, cuando queda claro por contexto, se puede usar directamente el símbolo $\models$ en lugar de $\models_T$.

Se describe a continuación la semántica formal de las fórmulas del lenguaje L. En todos los casos, p y q son fórmulas temporales:

1) $(\pi, i) \models \neg p \leftrightarrow (\pi, i) \not\models p$.
2) $(\pi, i) \models p \vee q \leftrightarrow (\pi, i) \models p \vee (\pi, i) \models q$.
3) $(\pi, i) \models \mathbb{X}p \leftrightarrow |\pi| > i + 1 \wedge (\pi, i + 1) \models p$.
4) $(\pi, i) \models \mathbb{Y}p \leftrightarrow i > 0 \wedge (\pi, i - 1) \models p$.
5) $(\pi, i) \models p \,\mathbb{U}\, q \leftrightarrow \exists k: i \leq k < |\pi| \wedge (\pi, k) \models q \wedge \forall m: i \leq m < k \rightarrow (\pi, m) \models p$.
6) $(\pi, i) \models p \,\mathbb{S}\, q \leftrightarrow \exists k: 0 \leq k \leq i \wedge (\pi, k) \models q \wedge \forall m: k < m \leq i \rightarrow (\pi, m) \models p$.

□

Para facilitar la escritura de las fórmulas de L, se pueden agregar los conectivos $\wedge$ y $\rightarrow$ (que se definen a partir de $\neg$ y $\vee$), y los siguientes operadores temporales (en todos los casos, p es una fórmula temporal):

– $\mathbb{F}p = \text{true}\,\mathbb{U}\,p$. La fórmula $\mathbb{F}p$ expresa que «alguna vez en el futuro» se cumplirá la propiedad p. El operador $\mathbb{F}$ (por *future*, es decir futuro) es un operador temporal de futuro.

– $\mathbb{G}p = \neg\,\mathbb{F}\,\neg\,p$. La fórmula $\mathbb{G}p$ expresa que no se cumple que alguna vez en el futuro no se cumplirá la propiedad p, o lo que es lo mismo, que se cumple «siempre» la propiedad p. El operador $\mathbb{G}$ (por *globally*, es decir globalmente) es un operador temporal de futuro.

– $\mathbb{O}p = \text{true}\,\mathbb{S}\,p$. La fórmula $\mathbb{O}p$ expresa que «alguna vez en el pasado» se cumplió la propiedad p. El operador $\mathbb{O}$ (por *once*, es decir una vez) es un operador temporal de pasado.

□

Por ejemplo, la fórmula $f = \mathbb{F}\,\neg\,\mathbb{X}\,\text{true}$, expresa que alguna vez en el futuro no habrá siguiente estado, es decir que $\pi \models f$ sii la computación π es finita.
Y la fórmula $f = \neg\,\mathbb{Y}\,\text{true}$, expresa que no existe estado anterior, es decir que para toda computación π se cumple $\pi \models f$. La fórmula f es equivalente a la expresión first mencionada anteriormente.
Podría trabajarse solamente con operadores de futuro sin alterar la expresividad de L. Por otro lado, aún considerando sólo el futuro, podría omitirse el operador $\mathbb{X}$, si no se pretendiera hacer referencia a una cantidad precisa de estados siguientes. Hay un ejercicio al final de esta parte del libro relacionado con el manejo preciso de unidades de tiempo.

A3.2.3. Método de verificación

Se presenta a continuación el método de verificación T que se va a utilizar. Se muestran solamente algunos axiomas y reglas representativos, dado el carácter introductorio de este capítulo.
Como todo método de verificación, T tiene dos partes:

a) Una parte general, en este caso de axiomas y reglas de la lógica temporal lineal (sobre el dominio de los números enteros). Esta parte, como en cualquier método, es independiente del lenguaje de programación. En los capítulos anteriores no se describió, dado que se asumió conocido el lenguaje de especificación Assn.

b) Una parte de programa. En este caso, los axiomas y reglas de la parte de programa se relacionan con los programas P de MCC.

A3.2.3.1. (Fragmento de la) parte general del método T

Se describen cinco axiomas y una regla, referidos a un fragmento de futuro del lenguaje L. En todos los casos, p y q son fórmulas temporales:

1) Axioma A1 o INS (instanciación temporal). Toda instanciación temporal de una fórmula de estado verdadera es un axioma. Es decir, si f es una fórmula de estado verdadera, Q es un símbolo de proposición de f, y p es una fórmula temporal, entonces se cumple $\models_T f[Q \mid p]$, tal que f [Q | p] denota la sustitución en f de Q por p. Por ejemplo, en la lógica clásica se cumple $\models ((x = y) \wedge (x = y \rightarrow Q)) \rightarrow Q$. Entonces, si p = $\mathbb{F}$q, vale $\models_T ((x = y) \wedge (x = y \rightarrow \mathbb{F}q)) \rightarrow \mathbb{F}q$.

2) Axioma A2. $\mathbb{G}(p \rightarrow q) \rightarrow (\mathbb{G}p \rightarrow \mathbb{G}q)$.

3) Axioma A3. $\mathbb{G}p \rightarrow p$.

4) Axioma A4. $\mathbb{G}p \rightarrow \mathbb{G}\mathbb{G}p$.

5) Axioma A5 (clausura). Si A es un axioma, entonces $\mathbb{G}$A también es un axioma.

6) Regla MP (modus ponens). $(p \wedge (p \rightarrow q)) \rightarrow q$.

□

Queda como ejercicio para el lector, probar la sensatez de este fragmento, es decir, que los axiomas A1 a A5 son verdaderos, y que la regla de modus ponens es sensata.
Los siguientes ejemplos muestran pruebas dentro del método T. La expresión $\vdash_T f$, denota que f es un teorema de T. Si queda claro por contexto, se puede utilizar directamente la expresión $\vdash f$.

Ejemplo. Se cumple $\vdash p \rightarrow \mathbb{F}p$. Se puede utilizar la equivalencia $(p \rightarrow \mathbb{F}p) = (p \rightarrow \neg\, \mathbb{G}\, \neg\, p)$:

(1) $\mathbb{G} \neg p \rightarrow \neg p$	(A3)
(2) $(\mathbb{G} \neg p \rightarrow \neg p) \rightarrow (p \rightarrow \neg\, \mathbb{G} \neg p)$	(INS)
(3) $(p \rightarrow \neg\, \mathbb{G} \neg p)$	(MP, 1, 2)

El axioma de clausura permite obtener en T axiomas de la forma $\mathbb{G}A$, siendo A un axioma. El teorema de generalización que se prueba en el ejemplo siguiente, permite también obtener en T teoremas de la forma $\mathbb{G}p$, siendo p un teorema.

Ejemplo. Se cumple que si $\vdash p$, entonces $\vdash \mathbb{G}p$. Se utilizará inducción sobre la longitud de la prueba.
Si p es un axioma, entonces se cumple $\vdash \mathbb{G}p$ por el axioma de clausura.
Si p no es un axioma, entonces p se obtuvo por modus ponens considerando los pasos:

(a) $\vdash q \rightarrow p$,
(b) $\vdash q$,

para alguna fórmula temporal q.

Por lo tanto, por la hipótesis inductiva se cumple:

(c) $\vdash G(q \rightarrow p)$,
(d) $\vdash \mathbb{G}q$.

Utilizando el axioma A2 y modus ponens considerando el paso (c), queda:

(e) $\vdash Gq \rightarrow \mathbb{G}p$.

Finalmente, por modus ponens considerando los pasos (d) y (e), se llega a:

(f) $\vdash \mathbb{G}p$.

A3.2.3.2. (Fragmento de la) parte de programa del método T

Se describen dos reglas. Para que sean representativas, la primera se relaciona con la familia de propiedades safety, y la segunda, con la familia de propiedades liveness:

(1) Regla R1. Dado el programa P :: [|| P_i] de MCC, la condición inicial θ, y el conjunto de transiciones T, siendo p y q fórmulas temporales:

$$\frac{\begin{array}{l}\text{P1. } \theta \rightarrow p \\ \text{P2. } \mathbb{G}(p \rightarrow q) \\ \text{P3. } \{p\}\ \tau\ \{p\}, \text{ para toda transición } \tau \in T\end{array}}{\mathbb{G}q}$$

La regla R1 establece que una manera de probar en una computación π que siempre vale q, es probar en π que siempre vale p (inicialmente y después de toda transición), y que cuando vale p también vale q.
De este modo p es un invariante (y q también). Una expresión de la forma $\mathbb{G}q$ denota una propiedad de tipo safety.
Una terna de la forma $\{p\}\ \tau\ \{q\}$ se denomina *condición de verificación* de la transición τ relativa a las fórmulas temporales p y q. Informalmente, la condición expresa que si en un estado de una computación se cumple p, entonces después de la ejecución de τ se cumple q en el estado siguiente. Formalmente:
Una transición $\tau \in T$ tiene asociada una *aserción* $\rho_\tau(X, X')$, que expresa la relación entre un estado σ y su sucesor σ' cuando se progresa a lo largo de τ. Las variables de X se refieren al estado σ, y las variables de X', al estado σ'. Por ejemplo, si la transición $\tau = (a_k, a_m)$ de un proceso P_i representa el comando: $y = 0 \rightarrow x := x + 1$, la aserción ρ_τ completa sería:

$$\rho_\tau = (c_i = a_k) \wedge (y = 0) \wedge (c'_i = a_m) \wedge (x' = x + 1) \wedge (\bigwedge_{z \in X} z' = z, \text{ para todo } z \neq x).$$

Así, una fórmula f ' se puede definir inductivamente a partir de f con relaciones como las siguientes:

- $p(x_1, x_2, \ldots, x_m)' = p(x'_1, x'_2, \ldots, x'_m)$, siendo p una fórmula de estado. Naturalmente, p en el estado siguiente se expresa como p en términos de las variables en el estado siguiente.

– $(\mathbb{Y}p)' = p$. Es decir, si en el estado siguiente se cumple que en el estado anterior se cumple p, entonces quiere decir que en el estado corriente se cumple p.
– $(p\ \mathbb{S}\ q)' = q' \vee ((p\ \mathbb{S}\ q) \wedge p')$. Es decir, si en el estado siguiente se cumple $p\ \mathbb{S}\ q$, entonces quiere decir que en el estado siguiente se cumple q, o bien en el estado corriente se cumple $p\ \mathbb{S}\ q$ y en el estado siguiente se cumple p.

Con las consideraciones anteriores, entonces la condición de verificación $\{p\}\ \tau\ \{q\}$ se puede definir por la expresión:

$$\mathbb{G}((\rho_\tau \wedge p) \rightarrow q').$$

A la implicación $(\rho_\tau \wedge p) \rightarrow q'$ se le aplica el operador $\mathbb{G}$, porque la condición de verificación debe cumplirse en todos los estados de una computación.

□

(2) Regla R2. Dado el programa P :: [|| P_i] de MCC, un conjunto parcialmente ordenado bien fundado (W, <), y siendo p, q y r fórmulas temporales:

$$\begin{array}{l} \text{P1. } \mathbb{G}(r \rightarrow (q \vee p)) \\ \text{P2. } \mathbb{G}(p \rightarrow w\ \varepsilon\ W) \\ \text{P3. } \mathbb{G}\ ((p \wedge w = a) \rightarrow \mathbb{F}(q \vee (p \wedge w < a))) \\ \hline \qquad\qquad \mathbb{G}(r \rightarrow \mathbb{F}q) \end{array}$$

La regla R2 establece que una manera de probar en una computación π que siempre en el futuro de r vale q, es probar en π que r siempre implica q ó p, y que p asegura que en el futuro vale q o bien sigue valiendo p y se decrementa el variante w de un conjunto parcialmente ordenado bien fundado (W, <).

La expresión $\mathbb{G}(r \rightarrow \mathbb{F}q)$ denota una propiedad de tipo liveness. Notar que si en particular se cumple que r = true, y $q = \neg\, \mathbb{X}$ true, entonces $\mathbb{G}(r \rightarrow \mathbb{F}q)$ queda reducida a: $\mathbb{GF} \neg\, \mathbb{X}$ true, que expresa la propiedad de terminación.

□

Se presenta a continuación un ejemplo de verificación en el método T aplicando la regla R1:

Ejemplo. Sea el programa P de MCC de la figura A3.2.4:

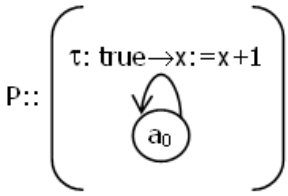


Figura A3.2.4

Dada $\theta = (x = 0 \wedge c = a_0)$, se va a probar que en la única computación π de P se cumple la propiedad safety $f = \mathbb{G}((x = 10) \rightarrow \mathbb{O}(x = 5))$, es decir, que siempre se cumple que si en el estado corriente vale x = 10, entonces alguna vez en el pasado se cumplió x = 5.
Se cumple $\pi \models_T f$: al comienzo vale x = 0, y después P, en cada transición, incrementa x en 1. Por lo tanto, si $\pi = \sigma_0, \sigma_1, \ldots, \sigma_k, \ldots$, entonces para todo i: $(\pi, i) \models_T (x = 10) \rightarrow \mathbb{O}(x = 5)$.

Para probar la fórmula f en T, se va a utilizar la regla R1, es decir:

P1. $\theta \rightarrow p$
P2. $\mathbb{G}(p \rightarrow q)$
P3. $\{p\}\ \tau\ \{p\}$, para toda transición $\tau \in T$
———————————————
$\mathbb{G}q$

donde $\theta = (x = 0 \wedge c = a_0)$, o lo que es lo mismo $\theta = (x = 0 \wedge first)$, y $q = ((x = 10) \rightarrow \mathbb{O}(x = 5))$.

Como invariante p se propone $p = ((x \geq 5) \rightarrow \mathbb{O}(x = 5))$.

Por lo tanto, la condición de verificación $\{p\}\ \tau\ \{p\}$ es:

$$\mathbb{G}\ (((x' = x + 1) \wedge ((x \geq 5) \rightarrow \mathbb{O}(x = 5))) \rightarrow ((x \geq 5) \rightarrow \mathbb{O}(x = 5))').$$

Prueba de P1:

$(x = 0 \wedge first) \rightarrow ((x \geq 5) \rightarrow \bigcirc(x = 5))$, se cumple por INST a partir de $(x = 0 \wedge first) \rightarrow ((x \geq 5) \rightarrow Q)$.

Prueba de P2:

$\Box (((x \geq 5) \rightarrow \bigcirc(x = 5)) \rightarrow ((x = 10) \rightarrow \bigcirc(x = 5)))$, se cumple por INST y el axioma de clausura a partir de $((x \geq 5) \rightarrow Q) \rightarrow ((x = 10) \rightarrow Q)$.

Prueba de P3:

Dado: $\Box (((x' = x + 1) \wedge ((x \geq 5) \rightarrow \bigcirc(x = 5))) \rightarrow ((x \geq 5) \rightarrow \bigcirc(x = 5))')$, como:
$((x \geq 5) \rightarrow \bigcirc(x = 5))' = (x' \geq 5) \rightarrow (\bigcirc(x = 5) \vee (x' = 5))$, queda:
$\Box (((x' = x + 1) \wedge ((x \geq 5) \rightarrow \bigcirc(x = 5))) \rightarrow ((x' \geq 5) \rightarrow (\bigcirc(x = 5) \vee (x' = 5))))$, que se cumple por INST y el axioma de clausura a partir de $((x' = x + 1) \wedge ((x \geq 5) \rightarrow Q)) \rightarrow ((x' \geq 5) \rightarrow (Q \vee (x' = 5)))$.

Finalmente, habiéndose probado las premisas P1 a P3, por la regla R1 de T se cumple:

$$\vdash_T \Box((x = 10) \rightarrow \bigcirc(x = 5)).$$

A3.2.4. Verificación automática y otras consideraciones

Cuando los dominios de las variables son finitos, se puede utilizar la lógica temporal proposicional. Las propiedades de las especificaciones Φ se construyen a partir de un conjunto AP de proposiciones atómicas, y la verificación de que un programa P satisface la especificación Φ se puede llevar a cabo automáticamente, lo que se conoce como *model checking*.
Los programas con estas características se denominan *programas de estados finitos*. Casos típicos son los algoritmos relacionados con protocolos de comunicación y diseños de circuitos.
La complejidad computacional del model checking depende del tipo de lógica temporal utilizada, lineal o ramificada. Entre ambas lógicas también existen diferencias de expresividad.
Un ejemplo de lenguaje de lógica temporal lineal proposicional es LTL. Una fórmula $\Phi \varepsilon$ LTL tiene la forma:

$\Phi :: T \mid F \mid p \mid (\neg \Phi) \mid (\Phi \vee \Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \rightarrow \Phi) \mid (\mathbb{X}\Phi) \mid (\Phi \,\mathbb{U}\, \Phi) \mid (\mathbb{G}\Phi) \mid (\mathbb{F}\Phi)$,

donde p es una proposición atómica de AP, y T y F son las constantes de LTL que representan los valores verdadero y falso, respectivamente. Como se aprecia, se utilizan solamente operadores temporales de futuro, y hay redundancia de conectivos lógicos y operadores temporales para facilitar la escritura de las fórmulas.

□

La semántica de LTL se define sobre diagramas de transiciones de estados que representan programas. Se consideran ternas de la forma:

$$M = < \Sigma , \rightarrow , L > .$$

M es una estructura compuesta por un diagrama de transiciones de estados y conjuntos de proposiciones atómicas asociados a los estados. La relación «$\rightarrow$» es una relación binaria que establece los sucesores de cada estado de Σ en las distintas computaciones o caminos de M. Y la función L: $\Sigma \rightarrow \mathcal{P}$ (AP) asigna a cada estado un conjunto de proposiciones atómicas de AP que se cumplen en el mismo.
Por ser LTL un lenguaje de la lógica temporal lineal, su semántica es la misma que se definió previamente. De todos modos, hay que agregar en la definición lo relacionado con las constantes y las proposiciones atómicas, entendiendo ahora, además, que las variables de X varían en un dominio finito. Se utiliza la notación M, $\pi \models \Phi$, para indicar que $\pi \varepsilon$ M. Por ejemplo:

1) M, $\pi \models$ T
2) M, $\pi \not\models$ F
3) M, $\pi \models$ p $\leftrightarrow$ p ε L(σ_0)
4) M, $\pi \models \neg \Phi \leftrightarrow$ M, $\pi \not\models \Phi$

- -

Ejercicio. *Completar la definición semántica del lenguaje LTL.*

□

Existen varios algoritmos de model checking sobre programas especificados con lenguajes como LTL, pero todos se basan en la misma estrategia general. Para verificar si M $\models \Phi$, es decir si se cumple Φ en todas las computaciones π de M:

1) Se construye un autómata A para la fórmula $\neg \Phi$, denominado $A_{\neg \Phi}$, que acepta todas las secuencias de valuaciones de las proposiciones atómicas que satisfacen $\neg \Phi$.
2) Se combina el autómata $A_{\neg \Phi}$ con la estructura M, lo que produce un diagrama de transiciones cuyos caminos son tanto de $A_{\neg \Phi}$ como de M.
3) Si no existe ningún camino en el diagrama resultante, el model checker responde que sí (vale $M \models \Phi$ porque ninguna computación π satisface $\neg \Phi$). En caso contrario responde que no.

Los lenguajes del tipo LTL tienen *fórmulas de camino*. Las fórmulas están implícitamente cuantificadas universalmente, por lo que no se pueden expresar propiedades sobre sólo algunas computaciones.
En este caso, la alternativa es recurrir a la lógica temporal ramificada. Los lenguajes de la lógica temporal ramificada, como por ejemplo CTL (por *Computation Tree Logic*), introducen explícitamente los cuantificadores universal y existencial. Una fórmula $\Phi \varepsilon$ CTL tiene la forma:

$\Phi :: T \mid F \mid p \mid (\neg \Phi) \mid (\Phi \vee \Phi) \mid (\Phi \wedge \Phi) \mid (\Phi \rightarrow \Phi) \mid (\mathbb{A}\mathbb{X}\Phi) \mid (\mathbb{E}\mathbb{X}\Phi) \mid (\mathbb{A}(\Phi \mathbb{U} \Phi)) \mid (\mathbb{E}(\Phi \mathbb{U} \Phi)) \mid (\mathbb{A}\mathbb{G}\Phi) \mid (\mathbb{E}\mathbb{G}\Phi) \mid (\mathbb{A}\mathbb{F}\Phi) \mid (\mathbb{E}\mathbb{F}\Phi)$,

donde p es una proposición atómica de AP, y T y F son las constantes que representan los valores verdadero y falso, respectivamente. $\mathbb{A}$ y $\mathbb{E}$ son, respectivamente, los cuantificadores universal y existencial.

□

Informalmente, la fórmula $\mathbb{A}f$ se cumple en un estado σ_0, si f se cumple en todas las computaciones a partir de σ_0. En cambio, la fórmula $\mathbb{E}f$ se cumple en σ_0, si f se cumple en alguna computación desde σ_0 (ver la figura A3.2.5).

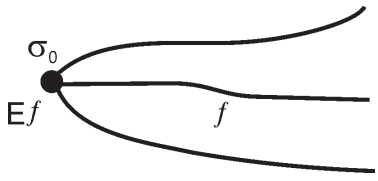


Figura A3.2.5

A diferencia de LTL, ahora el lenguaje CTL tiene *fórmulas de estado*. Entonces, dada la estructura $M = < \Sigma , \rightarrow , L >$, la semántica de una fórmula de CTL se define a partir de un estado σ, entendido como la raíz de un árbol de computaciones π . Se utiliza la notación $M, \sigma \models \Phi$. Para los casos sin cuantificación, se define:

1) $M, \sigma \models T$ y $M, \sigma \not\models F$
2) $M, \sigma \models p \leftrightarrow p \in L(\sigma)$
3) $M, \sigma \models \neg \Phi \leftrightarrow M, \sigma \not\models \Phi$

- -

y para las fórmulas cuantificadas, se define:

4) $M, \sigma \models AX\Phi$ sii para todo σ_1 tal que $\sigma \rightarrow \sigma_1$, se cumple $\sigma_1 \models \Phi$
5) $M, \sigma \models EX\Phi$ sii para algún σ_1 tal que $\sigma \rightarrow \sigma_1$ se cumple $\sigma_1 \models \Phi$
6) $M, \sigma \models AG\Phi$ sii para todos los estados σ_i de todas las computaciones $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \ldots$, tal que $\sigma = \sigma_1$, se cumple $M, \sigma_i \models \Phi$
7) $M, \sigma \models EG\Phi$ sii para todos los estados σ_i de alguna computación $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3 \rightarrow \ldots$, tal que $\sigma = \sigma_1$, se cumple $M, \sigma_i \models \Phi$

- -

Ejercicio. *Completar la definición semántica del lenguaje CTL.*

□

Para verificar programas especificados con lenguajes como CTL, los algoritmos típicos de model checking consisten, dada la estructura M, en etiquetar los estados de M que satisfacen Φ. Se empieza con «poblar» los estados con las subfórmulas más pequeñas de Φ, y luego iterativamente se tratan las subfórmulas de mayor longitud, considerando todos los casos de los operadores, hasta llegar a la fórmula definitiva Φ.

Los lenguajes CTL y LTL tienen expresividad incomparable, es decir, ninguno es más expresivo que el otro. CTL permite explícitamente cuantificar sobre computaciones, mientras que con LTL se puede seleccionar un rango de computaciones con determinadas características (por ejemplo, sólo las computaciones que sean fair). En general, se considera que la lógica temporal lineal es más fácil de usar. Como contrapartida, el model checking con CTL es más eficiente. Para reunir en un solo marco las dos lógicas temporales, la lineal y la ramificada, se suele trabajar con lenguajes como CTL*, que combina

la expresividad de LTL y CTL. Obviamente, el model checking en CTL* es menos eficiente comparativamente.
La sintaxis de CTL* se define, por lo tanto, en términos de fórmulas de estado y fórmulas de camino:

Fórmulas de estado:
Φ :: T | F | p | ($\neg \Phi$) | ($\Phi \vee \Phi$) | ($\Phi \wedge \Phi$) | ($\Phi \rightarrow \Phi$) | ($\mathrm{A}\vartheta$) | ($\mathrm{E}\vartheta$),
donde p es una proposición atómica y ϑ es una fórmula de camino.

Fórmulas de camino:
ϑ :: Φ | ($\neg \vartheta$) | ($\vartheta \vee \vartheta$) | ($\vartheta \wedge \vartheta$) | ($\vartheta \rightarrow \vartheta$) | ($\mathrm{X}\vartheta$) | ($\vartheta\ \mathrm{U}\ \vartheta$) | ($\mathrm{G}\vartheta$) | ($\mathrm{F}\vartheta$),
donde Φ es una fórmula de estado.

Así, si ϑ es una fórmula de camino, entonces $\mathrm{A}\vartheta$ y $\mathrm{E}\vartheta$ son fórmulas de estado. Y si Φ es una fórmula de estado, también es una fórmula de camino. El lenguaje CTL* se define como el conjunto de todas las fórmulas de estado especificadas previamente.
De esta manera, el lenguaje CTL no es sino el subconjunto de fórmulas de estado Φ de CTL* en que las subfórmulas de camino ϑ están restringidas a las formas $\mathrm{X}\Phi$, $\Phi\mathrm{U}\Phi$, $\mathrm{G}\Phi$ y $\mathrm{F}\Phi$: en CTL, los operadores son pares de la forma AX, EX, AG, EG, etc. Esto explica por qué en este lenguaje no puede expresarse el fairness, para lo que se requiere utilizar los pares FG ó GF.
Con respecto a la relación entre CTL* y LTL, el lenguaje LTL no es sino el subconjunto de fórmulas de estado Φ de CTL* de la forma $\mathrm{A}\vartheta$, tal que ϑ es una fórmula de camino sin ningún tipo de restricción: LTL se definió como un conjunto de fórmulas de camino implícitamente cuantificadas universalmente, en que se pueden combinar libremente los operadores X, U, G y F.

A3.3. Sintaxis y semántica de las fórmulas de correctitud

En este capítulo se describe con más detalle el lenguaje de las fórmulas de correctitud, que se denominará Form. Previamente no se trató, por ejemplo, la sintaxis y semántica de las expresiones enteras y booleanas ni de las aserciones.
Un segundo aporte de este capítulo es introducir muy suscintamente la *semántica denotacional*. Dado que anteriormente se utilizó en todos los casos la semántica operacional, se va a aprovechar esta ocasión para

emplear este tipo de semántica alternativa, mediante la cual también podría encararse el estudio de la verificación de programas.
A diferencia de la semántica operacional, que describe el comportamiento de un programa en términos de transiciones de configuraciones sobre una máquina abstracta, la semántica denotacional utiliza objetos matemáticos como conjuntos, funciones, funcionales, etc. Los programas se interpretan como funciones de estados iniciales a estados finales, sin tener en cuenta los estados intermedios (se trata de las funciones semánticas que se mencionaron junto a los lenguajes de programación considerados).
Otra característica de la semántica denotacional es el uso de distintas funciones para describir el significado de las construcciones del lenguaje, de manera tal que el significado o valor de una construcción se determina a partir del valor de sus componentes. La idea de construcciones del lenguaje que denotan valores, da el nombre a este tipo de semántica.
Para simplificar la presentación, se considerará solamente el lenguaje de programación PLW. Primero se describen denotacionalmente los lenguajes PLW, Assn y Form (Secciones A3.3.1 a A3.3.3). Luego se trata la problemática de la semántica denotacional de la instrucción while, y se hace referencia a otros lenguajes de programación (Secciones A3.3.4 y A3.3.5).

A3.3.1. Lenguaje PLW

A3.3.1.1. Sintaxis

La sintaxis de un programa de PLW se define a partir de los siguientes conjuntos:

a) Conjunto de símbolos terminales:

   { «;» , «,» , «:=» , «if» , «then» , «else» , «fi» , «while» , «do» , «od» , «+» , «=» , …}

b) Conjunto Icon de constantes enteras, con elementos denominados m, n, o, …

c) Conjunto Ivar de variables enteras, con elementos denominados x, y, z, …

d) Conjunto Iexp de expresiones enteras, con elementos denominados e_1, e_2, e_3, …

   e ε Iexp sii: e :: m | x | e_1 + e_2 | e_1 – e_2 | … | if B then e_1 else e_2 fi, con: m ε Icon, x ε Ivar, B ε Bexp (el conjunto Bexp se define enseguida).
   Ejemplos: 1, y, x + 8, if x > 0 then z + x else z – x fi.

e) Conjunto Bexp de expresiones booleanas, con elementos denominados B_1, B_2, B_3, …

   B ε Bexp sii: B :: true | false | e_1 = e_2 | e_1 < e_2 | ... | ¬B | B_1 → B_2, con: e_i ε Iexp.
   Ejemplos: true, x = y, ¬ (x < y), x – 1 > 0 → x > 0.

f) Conjunto PLW de instrucciones, con elementos denominados S_1, S_2, S_3, …

   S ε PLW sii: S :: skip | x := e | S_1 ; S_2 | if B then S_1 else S_2 fi | while B do S_1 od, con:
   x ε Ivar, e ε Iexp, B ε Bexp.
   Ejemplos: x := x + 1, if x > 0 then x := x – 1 else y := y – 1 fi, while true do skip od.

A3.3.1.2. Semántica

La semántica de un programa de PLW se define a partir de los siguientes conjuntos y funciones:

a) Conjunto V de los números enteros, con elementos denominados α_1, α_2, α_3, …

b) Conjunto W de los valores de verdad, o sea {tt, ff}, con elementos denominados β_1, β_2, β_3, …
   Los valores de verdad se nombran tt y ff, para diferenciarlos de los elementos sintácticos true y false que los representan, respectivamente.

c) Conjunto Σ de los estados, con elementos denominados σ_1, σ_2, σ_3, …
   Σ es el conjunto de todas las funciones de Ivar a V, es decir: Σ = (Ivar → V). Por lo tanto, si σ ε Σ, entonces σ: Ivar → V.

d) Asociadas a los operadores «+», «=», «¬», «→», …, se dispone de las funciones:
$+ : V \times V \to V$,
$= : V \times V \to W$,
$\neg : W \to W$,
$\to : W \times W \to W$,
- - - - - - - - - - - - -
con la semántica habitual.

e) La función $\mathcal{V}$: Iexp $\to (\Sigma \to V)$ define la semántica de las expresiones enteras:

$\mathcal{V}(m)(\sigma) = \alpha$, donde m es la representación del número entero α.

$\mathcal{V}(x)(\sigma) = \sigma(x)$.

$\mathcal{V}(e_1 + e_2)(\sigma) = \mathcal{V}(e_1)(\sigma) + \mathcal{V}(e_2)(\sigma)$, $\mathcal{V}(e_1 - e_2)(\sigma) = \mathcal{V}(e_1)(\sigma) - \mathcal{V}(e_2)(\sigma)$, …

$\mathcal{V}$(if B then e_1 else e_2 fi)(σ) = if $\mathcal{W}(B)(\sigma)$ then $\mathcal{V}(e_1)(\sigma)$ else $\mathcal{V}(e_2)(\sigma)$ fi, siendo:
la función $\mathcal{W}$ tal como se describe enseguida, y si $\alpha_1, \alpha_2 \in V$ y $\beta \in W$:
if β then α_1 else α_2 fi = α_1, si β = tt,
α_2, si β = ff.

f) La función $\mathcal{W}$: Bexp $\to (\Sigma \to W)$ define la semántica de las expresiones booleanas:

$\mathcal{W}$(true)(σ) = tt, y $\mathcal{W}$(false)(σ) = ff.

$\mathcal{W}(e_1 = e_2)(\sigma) = (\mathcal{V}(e_1)(\sigma) = \mathcal{V}(e_2)(\sigma))$, $\mathcal{W}(e_1 < e_2)(\sigma) = (\mathcal{V}(e_1)(\sigma) < \mathcal{V}(e_2)(\sigma))$, …

$\mathcal{W}(\neg B)(\sigma) = \neg\, \mathcal{W}(B)(\sigma)$, y $\mathcal{W}(B_1 \to B_2)(\sigma) = \mathcal{W}(B_1)(\sigma) \to \mathcal{W}(B_2)(\sigma)$.

g) Finalmente, la función $\mathcal{M}$: PLW $\to (\Sigma \to \Sigma)$ define la semántica de las instrucciones de PLW:

$\mathcal{M}$ (skip)$(\sigma) = \sigma$.

$\mathcal{M}$ (x := e)$(\sigma) = \sigma[x \mid \mathcal{V}(e)(\sigma)]$. Dado $\alpha \in V$:
$\sigma[x \mid \alpha](y) = \alpha$, si x = y,
$\sigma(y)$, si $x \neq y$.

$\mathcal{M}(S_1 ; S_2)(\sigma) = \mathcal{M}(S_2)(\mathcal{M}(S_1)(\sigma))$.

$\mathcal{M}$ (if B then S_1 else S_2 fi)(σ) = if $\mathcal{W}(B)(\sigma)$ then $\mathcal{M}(S_1)(\sigma)$ else $\mathcal{M}(S_2)(\sigma)$ fi. Dados $\sigma_1, \sigma_2 \in \Sigma$, y $\beta \in W$:
if β then σ_1 else σ_2 fi = σ_1, si β = tt,
σ_2, si β = ff.

$\mathcal{M}$ (while B do S od)(σ) = if $\mathcal{W}(B)(\sigma)$ then $\mathcal{M}$ (while B do S od)$(\mathcal{M}(S)(\sigma))$ else σ fi.

Como se aprecia, ahora se corrigen los abusos de notación σ(e) y σ(B) cometidos en un capítulo precedente. Las expresiones correctas son $\mathcal{V}$(e)(σ) y $\mathcal{W}$(B)(σ), respectivamente.

Con la función semántica $\mathcal{M}$ se pretende especificar denotacionalmente el comportamiento de los programas de PLW. Sin considerar la instrucción while, la existencia y unicidad de $\mathcal{M}$ se comprueban fácilmente: desde el punto de vista sintáctico, las instrucciones gozan de los atributos de *generación finita* y *única descomposición*, y la semántica de cada instrucción se define por inducción estructural a partir de la semántica de sus componentes. En el caso de la semántica del while, en cambio, habrá que resolver una ecuación recursiva con incógnita $\mathcal{M}$ (while B do S od), lo que amerita cierto desarrollo matemático que se presenta resumidamente en una sección posterior. En dicha sección se va a tratar también la inclusión del estado indefinido ⊥.

Antes de describir denotacionalmente los lenguajes Assn y Form, se presenta a continuación un par de ejemplos de aplicación de las definiciones formuladas:

Ejemplo. Si σ(x) = 3 y σ(y) = 4, entonces $\mathcal{V}$(if x = y then x + z else y + 1 fi)(σ) = 5:

$\mathcal{V}$(if x = y then x + z else y + 1 fi)(σ) = if $\mathcal{W}$(x = y)(σ) then $\mathcal{V}$(x + z)(σ) else $\mathcal{V}$(y + 1)(σ) fi.

Como: $\mathcal{W}$(x = y)(σ) = ($\mathcal{V}$(x)(σ) = $\mathcal{V}$(y)(σ)) = (σ(x) = σ(y)) = (3 = 4) = ff,

y: $\mathcal{V}$(y + 1)(σ) = $\mathcal{V}$(y)(σ) + $\mathcal{V}$(1)(σ) = σ(y) + σ(1) = 4 + 1 = 5,

entonces queda: $\mathcal{V}$(if x = y then x + z else y + 1 fi)(σ) = 5.

Por otro lado, se cumple que $\mathcal{M}$ (x := 0 ; y := x + 1)(σ) = σ[x | 0][y | 1], cualquiera sea σ:

$\mathcal{M}$ (x := 0 ; y := x + 1)(σ) = $\mathcal{M}$ (y := x + 1)($\mathcal{M}$ (x := 0)(σ)) = $\mathcal{M}$ (y := x + 1)(σ[x | $\mathcal{V}$(0)(σ)]) =

$\mathcal{M}$ (y := x + 1)(σ'), con σ' = σ[x | $\mathcal{V}$(0)(σ)] = σ[x | 0].

Luego: $\mathcal{M}$ (y := x + 1)(σ') = σ'[y | $\mathcal{V}$(x + 1)(σ')] = σ'[y | $\mathcal{V}$(x)(σ') + $\mathcal{V}$(1)(σ')] = σ'[y | σ'(x) + 1] = σ [x | 0] [y | σ[x | 0](x) + 1] = σ[x | 0][y | 0 + 1] = σ[x | 0][y | 1].

A3.3.2. Lenguaje Assn

A3.3.2.1. Sintaxis

Las aserciones del lenguaje Assn tienen las siguientes formas:

p :: true | false | $e_1 = e_2$ | $e_1 < e_2$ | … | $\neg$ p | $p_1 \rightarrow p_2$ | $\exists$x [p], con:
x ε Ivar, e_i ε Iexp.
Los elementos de Assn se denominan p, q, r, …
Ejemplos: false, $\neg$ (x + 1 = y + 1), $\exists$y [x = y + 1].

A3.3.2.2. Semántica

La función $\mathcal{T}$: Assn $\rightarrow$ ($\Sigma \rightarrow$ W) define la semántica de las aserciones:

$\mathcal{T}$(true)(σ) = tt, y $\mathcal{T}$(false)(σ) = ff.
$\mathcal{T}(e_1 = e_2)(\sigma) = (\mathcal{V}(e_1)(\sigma) = \mathcal{V}(e_2)(\sigma))$, $\mathcal{T}(e_1 < e_2)(\sigma) = (\mathcal{V}(e_1)(\sigma) < \mathcal{V}(e_2)(\sigma))$, …
$\mathcal{T}(\neg p)(\sigma) = \neg\, \mathcal{T}(p)(\sigma)$, y $\mathcal{T}(p_1 \rightarrow p_2)(\sigma) = \mathcal{T}(p_1)(\sigma) \rightarrow \mathcal{T}(p_2)(\sigma)$.
$\mathcal{T}(\exists x\,[p])(\sigma)$ = tt, si existe algún $\alpha \varepsilon$ V tal que $\mathcal{T}(p)(\sigma[x \mid \alpha]) = tt$,
ff, en caso contrario.

A3.3.3. Lenguaje Form

A3.3.3.1. Sintaxis

Las fórmulas del lenguaje Form tienen las siguientes formas:

f :: p | {p} S {q} | <p> S <q>, con:
p, q ε Assn, S ε PLW.
Los elementos de Form se denominan f_1, f_2, f_3, …
Ejemplos: x = X, {z = 0} skip {z = 0}, <true> while y < 0 do y := y + 1 od <y = 0>.

Notar que una aserción también es una fórmula de Form, lo que concuerda con cómo se describieron los métodos de verificación: un paso de una prueba puede ser una aserción verdadera del conjunto Tr, expresada en el lenguaje Assn, que luego será utilizada por la regla de consecuencia.

También podrían considerarse otras fórmulas, como por ejemplo $S_1 = S_2$, referida a la equivalencia funcional de dos programas, y $f_1 \wedge f_2$.

A3.3.3.2. Semántica

La función $\mathcal{F}$: Form $\rightarrow (\Sigma \rightarrow W)$ define la semántica de las fórmulas:

$\mathcal{F}(p)(\sigma) = \mathcal{T}(p)(\sigma)$.

$\mathcal{F}(\{p\}\ S\ \{q\})(\sigma) = (\mathcal{T}(p)(\sigma) \wedge \mathcal{M}(S)(\sigma) \neq \perp) \rightarrow \mathcal{T}(q)(\mathcal{M}(S)(\sigma))$.
$\perp$ es el estado indefinido. Se hacen más referencias a él en la próxima sección.

$\mathcal{F}(<p>S<q>)(\sigma) = \mathcal{T}(p)(\sigma) \rightarrow (\mathcal{M}(S)(\sigma) \neq \perp \wedge \mathcal{T}(q)(\mathcal{M}(S)(\sigma)))$.

Considerando las otras fórmulas mencionadas previamente, se define:

$\mathcal{F}(S_1 = S_2)(\sigma) = (\mathcal{M}(S_1)(\sigma) = \mathcal{M}(S_2)(\sigma))$.

$\mathcal{F}(f_1 \wedge f_2)(\sigma) = \mathcal{F}(f_1)(\sigma) \wedge \mathcal{F}(f_2)(\sigma)$.

Todas las fórmulas f utilizadas en la prueba de correctitud de un programa son verdaderas, es decir que cumplen:

$\mathcal{F}(f)(\sigma) = tt$.

Como esto vale para todo estado σ de Σ, se dice también que las fórmulas son *válidas*. Otra acepción de validez no utilizada en este libro, dado que se considera sólo el dominio de los números enteros, se refiere a las fórmulas verdaderas en todas las interpretaciones.

A3.3.4. Semántica denotacional de la instrucción while

En esta sección se trata la resolución de la ecuación recursiva que se planteó previamente, es decir:

$\mathcal{M}$(while B do S od)(σ) =
if $\mathcal{W}(B)(\sigma)$ then $\mathcal{M}$(while B do S od)($\mathcal{M}(S)(\sigma)$) else σ fi.

La función solución ϑ que se busca, entonces, expresará denotacionalmente la semántica de la instrucción while.
Si O(while B do S od)(σ) expresa operacionalmente la semántica del while, naturalmente $\vartheta(\sigma)$ deberá coincidir, cualquiera sea el estado σ, con O(while B do S od)(σ), que se definió antes de la siguiente manera:

a) σ, si $\mathcal{W}$(B)(σ) = ff.
b) σ', si existen $n \geq 1$ y estados $\sigma_0, \sigma_1, \ldots, \sigma_n$, con $\sigma = \sigma_0$, $\sigma' = \sigma_n$, $\sigma_i = O(S)(\sigma_{i-1})$, i = 1…n, siendo $\mathcal{W}(B)(\sigma_i) = tt$ cuando $i < n$, y $\mathcal{W}(B)(\sigma_i) = ff$ cuando $i = n$.
c) ⊥, en el caso restante.

La expresión ⊥ denota la no terminación del while (⊥ es un estado indefinido). Esto permite que la función O sea total. La función O también es *estricta*, es decir, O(⊥) = ⊥. Informalmente, la idea es que no puede obtenerse a partir de un estado indefinido un estado definido o propio. En otras palabras, una vez que un subprograma no termina, esta condición se extiende a todo el programa.
Para el cálculo de ϑ también se requerirá, por lo tanto, que la función sea total y estricta. Para ello se deben reformular algunas de las definiciones anteriores. Haciendo:

– $V' = V \cup \{\perp_V\}$, es decir, V' es el conjunto de los números enteros más el entero indefinido,
– $W' = W \cup \{\perp_W\}$, es decir, W' es el conjunto {tt, ff} más el valor de verdad indefinido,
– $\Sigma' = \Sigma \cup \{\perp_\Sigma\}$, es decir, Σ' es el conjunto de todos los estados más el estado indefinido ⊥,

se define:

– $\mathcal{V}$: Iexp → (Σ' → V'), con $\mathcal{V}(e)(\perp_\Sigma) = \perp_V$ para toda expresión entera e, y el resto como antes,
– $\mathcal{W}$: Bexp → (Σ' → W'), con $\mathcal{W}(B)(\perp_\Sigma) = \perp_W$ para toda expresión booleana B, y el resto como antes,
– $\mathcal{M}$: PLW → (Σ' → Σ'), con $\mathcal{M}(S)(\perp_\Sigma) = \perp_\Sigma$ para todo programa S, y el resto como antes,
– $\mathcal{T}$: Assn → (Σ' → W'), con $\mathcal{W}(p)(\perp_\Sigma) = ff$ para toda aserción p, y el resto como antes. A diferencia del caso de las expresiones booleanas, cuando se trata con aserciones tiene sentido referirse solamente a los valores de verdad tt ó ff.

Así, todas las funciones utilizadas son totales y estrictas. En este marco, y utlizando *órdenes parciales completos* (que se definirán enseguida), la función buscada ϑ se podrá obtener como el límite de una secuencia de funciones ϑ_i, con $i \geq 0$, donde cada ϑ_i, cuando $i \geq 1$, representa a lo sumo $i - 1$ ejecuciones del cuerpo S del while.
En lo que sigue, se presentan los pasos para calcular ϑ. Para facilitar la escritura, $\perp_\Sigma$ se escribirá directamente $\perp$, y Σ denotará el conjunto ampliado de estados propios más el $\perp$.

Definición. Una relación de orden parcial «$\leq$» sobre un conjunto C, denotada por $(C, \leq)$, es un orden parcial completo (se abrevia opc), sii:

a) El conjunto C tiene primer elemento, es decir, existe $x \in C$ tal que para todo $y \in C$, se cumple que $x \leq y$.
b) Toda cadena ascendente de C tiene cota superior mínima (se abrevia csm), es decir, dado $\{x_0, x_1, x_2, \ldots\} \subseteq C$ con $x_0 \leq x_1 \leq x_2 \leq \ldots$, entonces existe $y \in C$ que cumple $x_i \leq y$ para todo x_i, y si también existe $z \in C$ que cumple $x_i \leq z$ para todo x_i, entonces $y \leq z$.

Para facilitar la escritura, una cadena se entenderá siempre como una cadena ascendente, y la cadena $x_0 \leq x_1 \leq x_2 \leq \ldots$ se denotará con $\{x_i\}_{i \geq 0}$. Además, el opc $(C, \leq)$ podrá referirse directamente como el opc C cuando quede clara por contexto la relación «$\leq$».

□

Dado el conjunto C y un elemento fijo $x_0 \in C$, se define el opc *discreto* $(C, \leq_1)$ de la siguiente manera:

Para todo par de elementos y, z de C: $y \leq_1 z \leftrightarrow (y = x_0 \vee y = z)$.

Es decir que todo elemento de C se relaciona solamente con sí mismo, salvo el x_0 que se relaciona con todos y por lo tanto es el primer elemento.

Ejercicio. *Probar que C es un opc.*

□

Instanciando el conjunto C en Σ y el elemento fijo x_0 en $\perp$, se obtiene el opc discreto Σ. De una manera similar se obtienen los opc discretos V' y W'.
Sea ahora $D = (\Sigma \to \Sigma)$ el conjunto de todas las funciones de Σ a Σ. Denominando ϑ_i a las funciones de D, se va a definir la relación «$\leq_2$» entre las ϑ_i de la siguiente manera:

$\vartheta_i \leq_2 \vartheta'_i \leftrightarrow$ para todo $\sigma \in \Sigma$ se cumple que $\vartheta_i(\sigma) \leq_1 \vartheta'_i(\sigma)$.

Es decir que $\vartheta_i \leq_2 \vartheta'_i$ sii cualquiera sea el estado σ, $\vartheta_i(\sigma) = \perp$ o bien $\vartheta_i(\sigma) = \vartheta'_i(\sigma)$. En otras palabras, la función ϑ'_i está tan o más definida que la función ϑ_i, o lo mismo, ϑ_i es una aproximación de ϑ'_i.

Ejercicio. *Probar que D es un opc.*

□

Restringiendo D al conjunto D_e de las funciones estrictas, es decir las funciones ϑ_i que cumplen $\vartheta_i(\perp) = \perp$, sigue valiendo que D_e es un opc (queda como ejercicio para el lector).
Finalmente, en el marco del opc D_e, ya se está en condiciones de plantear cómo se calcula la función ϑ que expresa denotacionalmente la semántica del while. Haciendo:

a) $\vartheta_0(\sigma) = \perp$,
b) $\vartheta_{i+1}(\sigma)$ = if $\mathcal{W}$(B)(σ) then $\vartheta_i(\mathcal{M}(S)(\sigma))$ else σ fi, con $i \geq 0$, $\vartheta_i : \Sigma \rightarrow \Sigma$, S ≠ while B do S_1 od,

se prueba que todas las funciones ϑ_i son estrictas y que $\{\vartheta_i\}_{i \geq 0}$ es una cadena de D_e. Por lo tanto, existe ϑ = csm $\{\vartheta_i\}_{i \geq 0}$, que se obtiene como el límite, cuando $i \rightarrow \infty$, de la secuencia de ϑ_i.

Ejercicio. *Probar que todas las funciones ϑ_i son estrictas y que $\{\vartheta_i\}_{i \geq 0}$ es una cadena de D_e.*

□

La función ϑ es total, estricta, y cumple para todo estado σ que $\vartheta(\sigma)$ = O(while B do S od)(σ). La función ϑ es la solución buscada para el while. Cada ϑ_i, cuando $i \geq 1$, representa a lo sumo i – 1 ejecuciones del cuerpo S del while. La existencia y unicidad de la función ϑ se debe a que se calcula como la csm de una cadena de un opc.

Para cerrar esta sección, se presenta un ejemplo de cálculo de la función ϑ:

Ejemplo. Sea el programa S :: while x > 0 do x := x – 1 od, y el estado inicial σ[x | 2]. El estado final $\vartheta(\sigma[x \mid 2])$ se calcula de la siguiente manera:

$\vartheta_0(\sigma[x \mid 2]) = \perp$

$\vartheta_1(\sigma[x \mid 2]) =$ if $\mathcal{W}(x > 0)(\sigma[x \mid 2])$ then $\vartheta_0(\sigma[x \mid 1])$ else $\sigma[x \mid 2]$ fi $= \vartheta_0(\sigma[x \mid 1]) = \perp$

$\vartheta_2(\sigma[x \mid 2]) =$ if $\mathcal{W}(x > 0)(\sigma[x \mid 2])$ then $\vartheta_1(\sigma[x \mid 1])$ else $\sigma[x \mid 2]$ fi $= \vartheta_1(\sigma[x \mid 1]) =$
if $\mathcal{W}(x > 0)(\sigma[x \mid 1])$ then $\vartheta_0(\sigma[x \mid 0])$ else $\sigma[x \mid 1]$ fi $= \vartheta_0(\sigma[x \mid 0]) = \perp$

$\vartheta_3(\sigma[x \mid 2]) =$ if $\mathcal{W}(x > 0)(\sigma[x \mid 2])$ then $\vartheta_2(\sigma[x \mid 1])$ else $\sigma[x \mid 2]$ fi $= \vartheta_2(\sigma[x \mid 1]) =$
if $\mathcal{W}(x > 0)(\sigma[x \mid 1])$ then $\vartheta_1(\sigma[x \mid 0])$ else $\sigma[x \mid 1]$ fi $= \vartheta_1(\sigma[x \mid 0]) =$
if $\mathcal{W}(x > 0)(\sigma[x \mid 0])$ then $\vartheta_0(\sigma[x \mid -1])$ else $\sigma[x \mid 0]$ fi $= \sigma[x \mid 0]$

$\vartheta_4(\sigma[x \mid 2]) =$ if $\mathcal{W}(x > 0)(\sigma[x \mid 2])$ then $\vartheta_3(\sigma[x \mid 1])$ else $\sigma[x \mid 2]$ fi $= \vartheta_3(\sigma[x \mid 1]) =$
if $\mathcal{W}(x > 0)(\sigma[x \mid 1])$ then $\vartheta_2(\sigma[x \mid 0])$ else $\sigma[x \mid 1]$ fi $= \vartheta_2(\sigma[x \mid 0]) =$
if $\mathcal{W}(x > 0)(\sigma[x \mid 0])$ then $\vartheta_1(\sigma[x \mid -1])$ else $\sigma[x \mid 0]$ fi $= \sigma[x \mid 0]$

Se comprueba fácilmente que $\vartheta_i(\sigma[x \mid 2]) = \sigma[x \mid 0]$, para todo $i \geq 3$. Por lo tanto:

csm $\{\vartheta_i(\sigma[x \mid 2])\}_{i \geq 0} = \lim_{i \to \infty} \vartheta_i(\sigma[x \mid 2]) = \sigma[x \mid 0]$, y entonces:

$\mathcal{M}$(while $x > 0$ do $x := x - 1$ od) $(\sigma[x \mid 2]) = \sigma[x \mid 0]$.

A3.3.5. Caracterización de la semántica del while por el punto fijo mínimo y otras consideraciones

La semántica de la instrucción while se puede caracterizar alternativamente como el *punto fijo mínimo* de un funcional determinado. La utilización de los puntos fijos mínimos es esencial para definir denotacionalmente, entre otras cosas, la semántica de los procedimientos recursivos.

Definición. Dado un opc C, una función $f \varepsilon (C \to C)$, y un elemento $x \varepsilon C$,

a) x es un punto fijo de f sii $f(x) = x$.
b) x es el punto fijo mínimo de f (se abrevia μf) sii x es un punto fijo de f, y si se cumple también que $y \varepsilon C$ es un punto fijo de f, entonces $x \leq y$.

□

Volviendo a la ecuación recursiva anterior:

$\mathcal{M}$ (while B do S od)(σ) =
if $\mathcal{W}$(B)(σ) then $\mathcal{M}$ (while B do S od) ($\mathcal{M}$ (S)(σ)) else σ fi,

la alternativa entonces es resolverla calculando el punto fijo mínimo del funcional asociado a la ecuación, que se define naturalmente de la siguiente manera:

$\mathbb{F} : D_e \to D_e$, con $\mathbb{F}(\Psi)(\sigma)$ = if $\mathcal{W}$(B)(σ) then $\Psi(\mathcal{M}(S)(\sigma))$ else σ fi.

Es decir, la función Ψ es la incógnita $\mathcal{M}$ (while B do S od), y pertenece a D_e (funciones estrictas de Σ a Σ).

Se prueba que $\vartheta = \text{csm}\{\vartheta_i\}_{i \geq 0} = \mu\mathbb{F}$, considerando las relaciones y las funciones ϑ_i y ϑ que se definieron en la sección anterior. De este modo, la función ϑ cumple:

a) $\mathbb{F}(\vartheta) = \vartheta$.
b) Para todo $\Psi \varepsilon D_e$, si $\mathbb{F}(\Psi) = \Psi$, entonces $\vartheta \leq_2 \Psi$.

Ejercicio. *Probar que* $\vartheta = \mu F$.

□

En este caso, la existencia y unicidad de la función ϑ se debe a que se calcula como el punto fijo mínimo de un funcional *continuo* $\mathbb{F}$, porque se prueba que toda función continua tiene punto fijo mínimo. Dado un opc (C, ≤), una función f: C → C es continua si se cumple para toda cadena $x_0 \leq x_1 \leq x_2 \leq \ldots$, que $f(\text{csm}\{x_i\}_{i \geq 0}) = \text{csm}\{f(x_i)\}_{i \geq 0}$. Se prueba fácilmente que el funcional $\mathbb{F}$ es continuo.

En el caso del lenguaje de programación PROC considerado en un capítulo anterior, es decir el lenguaje PLW ampliado con procedimientos, la semántica denotacional de los procedimientos $proc_1$, …, $proc_n$, se define planteando un sistema de ecuaciones recursivas de la forma:

$\Psi_1 = \mathbb{F}_1(\Psi_1, \Psi_2, \ldots, \Psi_n)$
$\Psi_2 = \mathbb{F}_2(\Psi_1, \Psi_2, \ldots, \Psi_n)$
- - - - - - - - - - - - - - - -
$\Psi_n = \mathbb{F}_n(\Psi_1, \Psi_2, \ldots, \Psi_n)$

y calculando el punto fijo mínimo $<\vartheta_1, \vartheta_2, \ldots, \vartheta_n> = \mu<\mathbb{F}_1, \mathbb{F}_2, \ldots, \mathbb{F}_n>$, tal que la terna $<\mathbb{F}_i, \Psi_i, \vartheta_i>$ se asocia al procedimiento $proc_i$.
La función semántica $\mathcal{M}$ asociada a PROC se define de la siguiente manera. Sea:

$$\mathcal{M} : \text{PROC} \rightarrow (\text{Ent} \rightarrow (\Sigma \rightarrow \Sigma)).$$

Sea Ide_Proc el conjunto de todos los nombres de procedimientos. Y sea Ent (por entorno) el conjunto de todas las declaraciones de procedimientos, es decir:

$$\text{Ent} = (\text{Ide\_Proc} \rightarrow (\Sigma \rightarrow \Sigma)).$$

De esta manera, parte de la definición de $\mathcal{M}$ sería, dado $\Gamma \varepsilon$ Ent (queda como ejercicio para el lector completar la definición):

a) $\mathcal{M}(\text{call proc})(\Gamma)(\sigma) = \Gamma(\text{proc})(\sigma)$
b) $\mathcal{M}(\text{skip})(\Gamma)(\sigma) = \sigma$
c) $\mathcal{M}(x := e)(\Gamma)(\sigma) = \sigma[x \mid \mathcal{V}(e)(\sigma)]$
d) $\mathcal{M}(S_1 ; S_2)(\Gamma)(\sigma) = \mathcal{M}(S_2)(\Gamma)[(\mathcal{M}(S_1)(\Gamma)(\sigma)]$

- -

□

El uso de la semántica denotacional se hace mucho más complicado cuando se considera la programación no determinística, y más aún la programación paralela.

Por ejemplo, en el caso del lenguaje de programación GCL, la función semántica $\mathcal{M}$ asociada se define de la siguiente manera:

$$\mathcal{M} : \text{GCL} \rightarrow (\Sigma \rightarrow \mathcal{P}(\Sigma)).$$

Se utiliza el conjunto de partes $\mathcal{P}(\Sigma)$ porque el output de un programa S ε GCL, a partir de un estado inicial $\sigma \varepsilon \Sigma$, es un conjunto de estados finales.

Si IF denota $[\square_i B_i \rightarrow S_i]$, y DO denota $*[\square_i B_i \rightarrow S_i]$, se define:

$\mathcal{M}(\text{IF})(\sigma) = \{f\}$, si $\sigma \models \wedge_i \neg B_i$ (f es el estado de falla)
$\{\sigma' \mid \exists i \varepsilon \Delta: \sigma \models B_i \wedge \sigma' \varepsilon \mathcal{M}(S_i)(\sigma)\}$, si $\sigma \models \vee_i B_i$

$\mathcal{M}$ (DO)(σ) = {σ}, si σ |= $\wedge_i \neg B_i$

{σ' | ∃i ε Δ: σ |= $B_i \wedge \sigma'$ ε $\mathcal{M}$ (DO)($\mathcal{M}$ (S_i)(σ))}, si σ |= $\vee_i B_i$

□

Como $\mathcal{M}$ (S_i)(σ) es un conjunto de estados, expresiones como $\mathcal{M}$ (DO)($\mathcal{M}$ (S_i)(σ)) deben definirse de alguna manera. Dado el conjunto de estados τ, se define naturalmente:

$$\mathcal{M}(S)(\tau) = \cup_{\sigma \varepsilon \tau} \mathcal{M}(S)(\sigma).$$

$\mathcal{M}$ (DO)(σ) puede incluir el estado indefinido ⊥. Se establece que $\mathcal{M}$ (S)(⊥) = {⊥}, es decir que la indefinición (infinitud) se propaga. También debe considerarse el caso $\mathcal{M}$ (S)(f); se establece que la falla se propaga, es decir que $\mathcal{M}$ (S)(f) = {f} .

La semántica denotacional del DO se obtiene calculando el punto fijo mínimo de un determinado funcional continuo. A continuación se mencionan algunos pasos preparatorios para esto.
Asumiendo no determinismo acotado, el conjunto de partes $\mathcal{P}$ (Σ) se restringe al conjunto $\mathcal{T}$ de los conjuntos de estados τ que son:

a) finitos distintos del ∅, o bien
b) infinitos que incluyen el estado indefinido ⊥.

Se define el opc ($\mathcal{T}$, $\leq_{\mathcal{T}}$) siguiente. Dados dos conjuntos de estados τ_1 y τ_2 de $\mathcal{T}$, $\tau_1 \leq_{\mathcal{T}} \tau_2$ sii:

a) $\perp \varepsilon \tau_1 \wedge (\tau_1 - \{\perp\}) \subseteq \tau_2$, o bien
b) $\perp \notin \tau_1 \wedge \tau_1 = \tau_2$.

Este orden se denomina de *Egli-Milner*. Informalmente, la idea es que τ_2 está más definido que τ_1, dado que todos los elementos definidos de τ_1 están en τ_2 , y no puede ser que τ_2 tenga el ⊥ y τ_1 no (existen otros órdenes que se pueden utilizar, como el de *Hoare* y el de *Smyth*; hay un ejercicio entre los formulados al final de esta parte del libro, que relaciona los tres órdenes mencionados).
Si $E_e = (\Sigma \rightarrow \mathcal{T})$ es el conjunto de todas las funciones estrictas que van de Σ a $\mathcal{T}$, y se define en E_e que $f_1 \leq f_2 \leftrightarrow f_1(\sigma) \leq_{\mathcal{T}} f_2(\sigma)$ para todo σ ε Σ, se prueba que (E_e , ≤) es un opc.

Al igual que lo que se definió en el marco de los lenguajes PLW y PROC, el último paso es especificar un funcional continuo $\mathbb{F}$ sobre funciones ψ, que en este caso pertenecen a E_e, y calcular la semántica del DO como el punto fijo mínimo de $\mathbb{F}$.

Referencias bibliográficas de la Parte III

Los primeros métodos de verificación de programas fueron introducidos en forma independiente por Naur (Naur, 1966) y Floyd (Floyd, 1967). Dichos métodos se basaron en anotaciones sobre diagramas de flujo. Posteriormente, Hoare desarrolló métodos de verificación utilizando sistemas deductivos (Hoare, 1969). Se han elaborado después métodos de prueba basados en lógicas modales: (Salwicki, 1970), (Pratt, 1976) y (Constable, 1977). En (Jones, 1992) y (Lamport, 1993) se recorre históricamente la evolución de la verificación de programas.

Se ha hecho referencia a los sistemas deductivos en distintas partes del presente libro, y en particular, al teorema de incompletitud de Gödel (en 1931 Gödel demostró la incompletitud de cualquier sistema deductivo consistente, que fuese lo suficientemente expresivo para incluir a la aritmética de los números naturales). Para un estudio introductorio a los sistemas deductivos recomendamos (Hamilton, 1991) y (Gries y Schneider, 1993). El método que utilizamos para definir la semántica operacional de los lenguajes de programación se describe en (Plotkin, 1981). La aproximación denotacional puede encontrarse, por ejemplo, en (de Bakker, 1980). En (Winskel, 1993) se comparan los dos tipos de semántica de una manera introductoria. Hay suficiente material de consulta para incursionar en la problemática de la especificación de programas, que no fue tratada por estar fuera del alcance del libro. Por ejemplo, recomendamos (Jackson, 1995), donde se discuten principios y técnicas generales para el análisis de requerimientos, la especificación y el diseño.

Axiomas para la asignación diferentes al utilizado en el libro fueron definidos en (Hoare y Wirth, 1973), (Gries, 1978) y (Apt, 1981). Las reglas de prueba redundantes AND y OR fueron presentadas en (Zwiers, 1989). En (Apt, 1981) se presentan distintas construcciones de programas. Se ha probado la existencia de construcciones de programas que no poseen reglas de Hoare sensatas y completas (Clarke, 1979-a). Se demuestra la sensatez y la completitud de las reglas como teoremas en la teoría de funciones, y es posible demostrar la imposibilidad de definir reglas de Hoare sensatas y completas para determinadas construcciones probando la inexistencia de puntos fijos. En (Reynolds, 1998) se utiliza la semántica denotacional para demostrar la sensatez y completitud de los métodos de verificación. La idea de proof outline se encuentra en (Constable y O'Donnel, 1978). En (Hoare, 1971) se presenta la verificación de programas con procedimientos.

La verificación de programas no determinísticos fue introducida en (Dijkstra, 1975). Su análisis y varias semánticas son analizadas en (de Bakker, 1980) y en (Apt, 1984). Los programas no determinísticos con

asignación aleatoria se estudian en (Apt y Plotkin, 1986). El concepto de no determinismo acotado se discutió inicialmente en (Dijkstra, 1976). En (Francez, 1986) se desarrolla ampliamente el concepto de fairness. Formas alternativas de fairness pueden encontrarse en (Alur y Henzinger, 1994) y (Lamport, 2000). La verificación de programas con fairness fue presentada fundamentalmente en (Apt y Olderog, 1983). Otros métodos fueron propuestos en forma independiente en (Lehmann, Pnueli y Stavi, 1981) y en (Grumberg, Francez, Makowsky y de Roever, 1985).

El método utilizado para la verificación de la correctitud parcial y total de los programas concurrentes con variables compartidas se debe a Owicki y Gries (Owicki y Gries, 1976-a). En (Hoare, 1972) ya se trataron las secciones críticas condicionales, y en (Owicki y Gries, 1976-b) se definen reglas para la verificación de programas concurrentes que las emplean. Una técnica de prueba similar fue introducida en forma independiente en (Lamport, 1977). El método de Owicki y Gries fue criticado por la pérdida de la composicionalidad, y así se han estudiado otros métodos (Boer, 1994). Semánticas alternativas a la de intercalación (*interleaving*) fueron tratadas en (Salwicki y Müldner, 1981). Otra alternativa en este contexto son las semánticas basadas en órdenes parciales sobre las configuraciones (Best, 1996). El concepto de monitor para encapsular datos y su manipulación concurrente se introdujo en (Hoare, 1974).

El lenguaje CSP fue definido por Hoare (Hoare, 1978). La primera semántica de CSP se encuentra en (Francez, Hoare, Lehmann y de Roever, 1979). La semántica operacional del lenguaje está en (Plotkin, 1982). Sistemas de prueba para la verificación de programas de CSP fueron introducidos en (Apt, Francez y de Roever, 1980) y en (Levin y Gries, 1981). Un repaso de varios sistemas de prueba para CSP aparece en (Hooman y de Roever, 1986). En (Francez, 1989) se presenta una extensión del método AFR para las *multiparty interactions*. El método de prueba de la terminación de los programas de CSP se presenta en (Grumberg, Francez y Katz, 1984). El tratamiento de la comunicación asincrónica se puede leer en (Schlichting y Schneider, 1984).

Los métodos de prueba descriptos en el presente libro se consideran excelentes guías para el desarrollo sistemático de programas. La problemática de la construcción sistemática de programas correctos se introduce explícitamente en (Dijkstra, 1976), aunque en esos años fueron numerosos los artículos y libros escritos simultáneamente sobre el tema, todos centrados en lo que se denominó programación estructurada. Por ejemplo, recomendamos de esa época (Dahl, Dijkstra y Hoare, 1974). Una continuación del trabajo de Dijkstra, aún en el marco de la programación secuencial, es (Gries, 1981). Un libro fundacional sobre

el desarrollo sistemático de programas concurrentes es (Chandy y Misra, 1988). Los autores proponen derivar programas concurrentes a partir de sus especificaciones. Se utiliza un lenguaje de programación muy simple denominado UNITY. Como novedad, utilizan una variante de la lógica temporal. También Lengauer propuso un método para la construcción de programas concurrentes, pero a partir de programas secuenciales (Lengauer, 1993). Distintos métodos para la derivación sistemática de programas concurrentes pueden encontrarse en (Knapp, 1992). En (Veloso, 1986) se plantea una interesante clasificación de métodos de construcción de programas, en que se comparan paradigmas basados en los trabajos, entre otros, (Jackson, 1975), (Dijkstra, 1976), (Goguen, Thatcher y Wagner, 1978), (Yourdon y Constantine, 1979) y (Bauer, 1982). En (Haeberer, Veloso y Baum, 1988) se plantea un formalismo para modelizar el proceso completo de desarrollo de software. Más en relación con la algorítmica y la educación, también recomendamos leer (Scholl y Peyrin, 1989), que es un libro de una saga en que se desarrolla una metodología de construcción de programas basada en máquinas de símbolos y una tipología de estructuras de iteración.

Pnueli introdujo la lógica temporal para la prueba de propiedades de tipo liveness sobre programas concurrentes (Pnueli, 1977). Un método para probar propiedades liveness sin lógica temporal había sido propuesto con anterioridad (Lamport, 1977). Está aceptado universalmente que la utilización de la lógica temporal permite expresar y probar propiedades de programas de una manera natural y simple. Una buena introducción a la lógica temporal es (Huth y Ryan, 2004). Las primeras definiciones de propiedades safety y liveness fueron dadas en (Lamport, 1977). Luego, Alpern y Schneider precisaron las definiciones informales y probaron que toda fórmula temporal puede ser expresada como una conjunción de una propiedad safety y una propiedad liveness (Alpern y Schneider, 1985).

La verificación de propiedades temporales restringida a programas con estados finitos es decidible (Pnueli, 1977). Se ha intentado encontrar algoritmos eficientes, como (Lichtenstein, Pnueli y Stavi, 1981), (Quelle y Sifakis, 1983) y (Clarke, Emerson y Sistla, 1986), y cotas inferiores de complejidad (Sistla y Clarke, 1985). Estos algoritmos recibieron la denominación de algoritmos de *model checking*, dado que la verificación consiste en comprobar que el programa es un modelo de la fórmula temporal considerada. En (Vardi, 2002) se analizan y comparan los algoritmos de model checking basados en los lenguajes LTL y CTL. En (Pnueli y Kesten, 2002) se presenta un sistema deductivo sensato y relativamente completo para CTL*. En (Schuppan y Biere, 2005) se

analizan algunos algoritmos de model checking con foco en la longitud de los contraejemplos obtenidos. La verificación de propiedades safety puede ser reducida a computar el conjunto de estados alcanzables de un sistema (Kupferman y Vardi, 2001). En (Kesten, Pnueli, Raviv y Shahar, 2006) se presenta un método de model checking para LTL teniendo en cuenta restricciones de fairness débil y fuerte. El trabajo (Vardi, 2007) puede ser consultado para incursionar en model checking utilizando la teoría de autómatas, y como resumen de los resultados obtenidos recientemente en dicha área.

A continuación se lista la que consideramos bibliografía principal en relación a la tercera parte del libro. En este sentido, (Francez, 1992) y (Apt y Olderog, 1997) tratan en conjunto varios de los temas que hemos desarrollado:

- Alpern B. y Schneider F. B. [1985]. Defining liveness. Information Processing Letters, 21(4):181–185.
- Alur R. y Henzinger T. A. [1994]. Finitary fairness, in: Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science (LICS '94), IEEE Computer Society Press, pp. 52-61.
- Apt K. R. [1981]. Ten years of Hoare's logic: a survey - Part I. ACM Trans. on Programming Languages and Systems 3, pp. 431-483.
- Apt K. R. [1984]. Then years of Hoare's logic: a survey - Part II. Theoretical Computer Science 28, pp. 83-109.
- Apt K. R., Bergstra J. A. y Meertens L. G. L. T. [1979]. Recursive assertions are not enough or are they? Theoret. Comp. Sci. 8, 73-87.
- Apt K. R., Francez N. y P. de Roever W. [1980]. A proof system for communicating sequential processes. ACM Trans. Prog. Lang. Syst., 2, pp. 359-385.
- Apt K. R. y Olderog E. R. [1983]. Proof rules and transformations dealing with fairness. Sci. Comp. Programming, 3, pp. 65-100.
- Apt K. R. y Olderog E. R. [1997]. Verification of sequential and concurrent programs. Springer, second edition.
- Apt K. R. y Plotkin G. D. [1986]. Countable nondeterminism and random assignment. J. Assoc. Comput. Mach., 33, pp. 724-767.
- Ashcroft E. y Manna Z. [1971]. Formalization of properties of parallel programs, Machine Intelligence, 6, pp. 17-41.
- Bauer F. L., Wössner H. [1982]. Algorithmic Language and Program Development. Springer-Verlag.
- Best E. [1996]. Semantics of Sequential and Parallel Programs, Prentice-Hall International, London.
- Blass A. y Gurevich Y. [2001]. Inadequacy of Computable Loop Invariants. ACM Transactions on Computational Logic Vol. 2:1, pp. 1-11.
- Boer F. S. [1994]. Compositionality in the inductive assertion method for concurrent systems, in: Programming Concepts, Methods and Calculi, E. R. Olderog, ed., Elsevier/North-Holland, Amsterdam, pp. 289-305.

- Chandy K. M. y Misra J. [1988]. Parallel Program Design: A Fundation. Addison-Wesley, New York.
- Clarke E. M. [1979-a]. Programming languages constructs for which it is impossible to obtain good Hoare axiom systems. Journal of the ACM, Vol 26, No 1, pp 129-147.
- Clarke E. M. [1979-b]. Program Invariants as Fixedpoints. Computing 21, pp 273-294.
- Clarke E. M., Emerson E. A. y Sistla A. P. [1986]. Automatic verification of finite-state concurrent systems using temporal logic specification. ACM Trans. on Programming Languages and Systems 8, pp. 244-263.
- Constable R. L. [1977]. On the theory of programming logics. Proc. 9th Symp. Theory of Comput. ACM, pages 269–285.
- Constable R. L. y O'Donnel M. [1978]. A Programming Logic. Wintrop.
- Dahl O. J., Dijkstra E. W., Hoare C. A. R. [1974]. Structured Programming. Academic Press, New York.
- de Bakker J. W. [1980]. Mathematical Theory of Program Correctness. Englewood Cliffs NJ, Prentice-Hall.
- Dijkstra E. W. [1968]. Cooperating sequential processes, Programming Languages: NATO Advanced Study Institute, F. Genuys, ed., Academic Press, London, pp. 43-112.
- Dijkstra E. W. [1975]. Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM, 18, pp. 453-457.
- Dijkstra E. W. [1976]. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N. J.
- Flon L. y Suzuki N. [1981]. The total correctness of parallel programs, SIAM J. Comput., pp. 227-246.
- Floyd R. W. [1967]. Assigning meanings to programs. Proceedings of the Symposium on Applied Math., Vol. 19, pages 19–32. American Mathematical Society.
- Francez N. [1986]. Fairness, Springer-Verlag, New York.
- Francez N. [1989]. Cooperating proofs for distributed programs with multiparty interactions. Information Processing Lett., 32, 235-42.
- Francez N. [1992]. Program verification. Addison-Wesley.
- Francez N., Hoare C. A. R., Lehmann D. J. y P. de Roever W. [1979]. Semantics of nondeterminism, concurrency and communications, J. Comput. System Sci., 19, pp. 290-308.
- Goguen J. A., Thatcher J. W. y Wagner E. G. [1978]. An initial algebra approach to the specification, correctness and implementation of abstract data types. Current trends in programming methodology, 81-149. Prentice-Hall.
- Gries D. [1978]. The multiple assignment statement, IEEE Trans. Softw. Eng., SE-4, pp. 89-93.
- Gries D. [1981]. The Science of Programming, Springer-Verlag, New York.
- Gries D. y Schneider F. B. [1993]. A Logical Approach to Discrete Math. Springer-Verlag, New York.

- Grumberg O., Francez N. y Katz S. [1984]. Fair termination of communicating processes. Proc. 3rd. ACM Annual Sym. On Principles of Distributed Computing, Vancouver.
- Grumberg O., Francez N., Makowsky J. A. y P. de Roever W. [1985]. A proof rule for fair termination of guarded commands, Information and Control, 66, pp. 83-102.
- Haeberer A., Veloso P. y Baum G. [1988]. Formalización del Proceso de Desarrollo de Software. Kapeluz.
- Hamilton A. G. [1991]. Logic for Mathematicians. Cambridge University Press.
- Hoare C. A. R. [1969]. An axiomatic basis for computer programming. Comm. ACM, 12, pp. 576-580.
- Hoare C. A. R. [1971]. Procedures and parameters: an axiomatic approach. Proc. Sym. On Semantics of Algorithmic Languages. Lecture Notes in Mathematics Vol. 188. Berlin: Springer.
- Hoare C. A. R. [1972]. Towards a theory of parallel programming, in: Operating Systems Techniques, C. A. R., R. H. Perrot, eds., Academic Press, London, pp. 61-71.
- Hoare C. A. R. [1974]. Monitors: an operating system structuring concept. Comm. ACM, 17, 549-57.
- Hoare C. A. R. [1978]. Communicating sequential processes. Comm. ACM, Vol 21, 8, pp 666-677.
- Hoare C. A. R. y Wirth N. [1973]. An axiomatic definition of the programming language Pascal, Acta Inf., 2, pp. 335-355.
- Hooman J. y P. de Roever W. [1986]. The quest goes on: a survey of proof systems for partial correctness of CSP. Current Trends in Concurrency, Lecture Notes in Computer Science 224, Springer-Verlag, New York, pp. 343-395.
- Huth M. y Ryan M. [2004]. Logic in Computer Science. Cambridge University Press.
- Jackson M. [1975]. Principles of Program Design. Academic Press, London.
- Jackson M. [1995]. Software Requirements & Specifications. Addison-Wesley.
- Jones C. B. [1992]. The search for tractable ways of reasoning about programs. Tech. Rep. UMCS-92-4-4. Department of Computer Science, University of Manchester, Manchester.
- Kesten Y. y Pnueli A. [2005]. A compositional approach to CTL* verification. Theoretical Computer Science, v.331 n.2-3, p.397-428.
- Kesten Y., Pnueli A., Raviv L. y Shahar E. [2006]. Model Checking with Strong Fairness. Formal Methods in System Design Vol 28:1, pp. 57-84.
- Knapp E. [1992]. Derivation of concurrent programs: two examples, Sci. Comput. Programming, 19, pp. 1-23.
- Kupferman O. y Vardi M. [2001]. Model checking of safety properties, Formal Methods in System Design 19, pp. 291-314.
- Lamport L. [1977]. Proving the correctness of multiprocess programs, IEEE Trans. Softw. Eng., SE-3:2, pp. 125-143.
- Lamport L. [1983]. What good is temporal logic? R. E. A. Mason, editor, Information Processing 83: Proceedings of the IFIP 9th World Congress, pages 657-668, Paris.

- Lamport L. [1993]. Verification and Specification of Concurrent Programs. Proceedings of a REX Workshop held, The Netherlands.
- Lamport L. [2000]. Fairness and Hyperfairness. Distrib. Comput. 13: 239-245, Springer-Verlag.
- Lehmann D. J., Pnueli A. y Stavi J. [1981]. Impartiality, justice, and fairness: the ethics of concurrent termination. Proceedings of International Colloquium on Automata Languages and programming (ICALP '81). Lecture Notes in Computer Science 115, Springer-Verlag, New York, pp. 264-277.
- Lengauer C. [1993]. Loop parallelization in the polytope model. CONCUR'93, E. Best, ed., Lecture Notes in Computer Science 715, Springer-Verlag, New York, pp. 398-416.
- Levin G. y Gries D. [1981]. A proof technique for communicating sequential processes, Acta Inf., 15, pp. 281-302.
- Lichtenstein O., Pnueli A. y Stavi J. [1981]. Impartiality, justice, and fairness - the ethic of concurrent termination. Proc. 8th Int'l Colloq. on Automata, Languague, and Programming, LNCS 115, Springer-Verlag, pp. 264-277.
- Loeckx J. y Sieber K. [1987]. The Foundation of Program Verification, second ed., Teubner.Wiley, Stuttgart.
- Manna Z. y Pnueli A. [1991]. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, New York.
- Naur P. [1966]. Proof of Algorithms by General Snapshots. BIT, 6, 310-316.
- Owicki S. y Gries D. [1976-a]. An axiomatic proof technique for parallel programs, Acta Inf., 6, pp. 319-340.
- Owicki S. y Gries D. [1976-b]. Verifying properties of parallel programs: an axiomatic approach, Comm. ACM, 19, pp. 279-285.
- Plotkin G. D. [1981]. A structural Approach to Operational Semantics. Technical Report DAIMI-FN 19, Computer Science Department, Aarhus University.
- Plotkin G. D. [1982]. An operational semantics for CSP, Formal Description of Programming Concepts II, D. Bjorner, ed., North-Holland, Amsterdam, pp. 199-225.
- Pnueli A. [1977]. The temporal logic of programs. Proceedings of the 18th Annual Symposium on the Foundations of Computer Science, pp 46-57. IEEE.
- Pnueli A. y Kesten Y. [2002]. A Deductive Proof System for CTL*. Proceedings of the 13th International Conference on Concurrency Theory. Lecture Notes In Computer Science; Vol. 2421, pp. 24-40.
- Pratt V. R. [1976]. Semantical considerations on Floyd-Hoare logic. Proc. 17th Symp. Found. Comput. Sci., pages 109-121. IEEE.
- Quelle J. P. y Sifakis J. [1983]. Fairness and related properties in transition systems - a temporal logic to deal with fairness. Acta Informática 19, pp. 195-220.
- Reynolds J. C. [1998]. Theories of Programming Languages. Cambridge University Press.
- Salwicki A. [1970]. Formalized algorithmic languages. Bull. Acad. Polon. Sci. Ser. Sci. Math. Astron. Phys., 18:227-232.

- Salwicki A. y Müldner T. [1981]. On the algorithmic properties of concurrent programs. Proc. of Logics of Programs, E. Engeier, ed., Lecture Notes in Computer Science 125, Springer-Verlag, New York, pp. 169-197.
- Schlichting R. D. y Schneider F. B. [1984]. Message passing for distributed programming. ACM Trans. Programming Languages Systems, 6, 402-31.
- Scholl P. C. y Peyrin J. P. [1989]. Schémas algorithmiques fondamentaux: séquences et itérations. Masson.
- Schuppan V. y Biere A. [2005]. Shortest Counterexamples for Symbolic Model Checking of LTL with Past. TACAS 2005, Edinburgh, UK.
- Sistla A. P. y Clarke E. M. [1985]. The complexity of propositional linear temporal logics. J. ACM 32, PP. 733-749.
- Vardi M. Y. [2002]. Branching vs. Linear Time: Final Showdown. Journal of Automata, Languages and Combinatorics 7(2): 225-246.
- Vardi M. Y. [2007]. Automata-Theoretic Model Checking Revisited. Proc. 7th Int'l Conf. on Verification, Model Checking, and Abstract Interpretation, Lecture Notes in Computer Science, 4349 (2007): 137-150.
- Veloso P. [1986]. Verificacao e construcao de programas. Editora da Unicamp.
- Winskel G. [1993]. The Formal Semantics of Programming Languages. An Introduction. The MIT Press, Cambridge, Massachusetts, London, England.
- Yourdon E. y Constantine L. L. [1979]. Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Prentice-Hall, Englewood Cliffs.
- Zwiers J. [1989]. Compositionality, Concurrency and Partial Correctness-Proof Theories for Networks of Processes and Their Relationship, Lecture Notes in Computer Science 321, Springer-Verlag, New York.

Ejercicios de la Parte III

E.III.1 Especificar un programa P que calcule en la variable y, la raíz cuadrada entera de un número natural x. La especificación debe establecer que P preserve el valor de x. ¿Se podría expresar además que x no se altere a lo largo de P?

E.III.2 Probar que para todo estado σ, se cumple que $\sigma[x \mid a][x \mid b] = \sigma[x \mid b]$.

E.III.3 Supóngase que la expresión $\sigma \models \{p\}\ S\ \{q\}$ denota la correctitud parcial de S con respecto al par <p, q> en el estado específico σ. Determinar si se cumple $\sigma \models \{p\}\ S\ \{q\}$ en los siguientes incisos:

(a) $\sigma \not\models p$ y $val(\pi(S, \sigma)) = \perp$ (b) $\sigma \models p$ y $val(\pi(S, \sigma)) = \perp$

E.III.4 Supóngase que la expresión $\sigma \models \langle p\rangle\ S\ \langle q\rangle$ denota la correctitud total de S con respecto al par <p, q> en el estado específico σ. Determinar si se cumple $\sigma \models \langle p\rangle\ S\ \langle q\rangle$ en los mismos incisos del ejercicio anterior.

E.III.5 La expresión var(p) denota el conjunto de variables libres de la aserción p de Assn, y la expresión var(S) denota el conjunto de las variables que aparecen en el programa S de PLW. Definir por inducción estructural los conjuntos var(p) y var(S).

E.III.6 Dado $S \in PLW$, probar que $\models \{p\}\ S\ \{q\} \leftrightarrow post(p, S) \subseteq \{\sigma \mid \sigma \models q\}$.

E.III.7 Probar que el axioma de asignación «hacia adelante»: $\{p\}\ x := e\ \{\exists z: p^x_z \wedge x = e^x_z\}$, planteado en el libro, es redundante en H.

E.III.8 Supóngase que se le agrega al lenguaje PLW la instrucción if B then S fi, que se define como una instrucción equivalente a la instrucción if B then S else skip fi. Proponer una regla de prueba para la nueva instrucción, que preserve la sensatez y la completitud relativa del método H.

E.III.9 Proponer una regla de prueba alternativa, en el método H, para la instrucción while, cuya conclusión sea {p} while B do S od {q}. Probar que la regla preserva la sensatez y la completitud relativa de H.

E.III.10 Probar la sensatez de las siguientes reglas, en el marco del método H:

$$\frac{\{p_1\}\ S\ \{q_1\}\ ,\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}} \qquad \frac{\{p_1\}\ S\ \{q_1\}\ ,\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \vee p_2\}\ S\ \{q_1 \vee q_2\}}$$

E.III.11 Probar la sensatez de la siguiente regla (regla SHIFT), en el marco del método H:

$$\frac{\{p \wedge q\}\ S\ \{r\},\ var(q) \cap var(S) = \varnothing}{\{p\}\ S\ \{q \rightarrow r\}}$$

E.III.12 Probar que las reglas de prueba AND y OR, planteadas en el libro para ampliar el método H, son redundantes.

E.III.13 Probar que para todo programa S de PLW y para toda aserción q de Assn, se cumple

$$Tr \vdash_H \{false\}\ S\ \{q\}$$

E.III.14 Determinar si se cumplen las siguientes fórmulas de correctitud, probándolas en H o H* en caso afirmativo, o mostrando un contraejemplo en caso negativo:

(a) {true} x := 0 {false}
(b) {y = 20} x := 4 {y = 20}
(c) {x = 0} while z = 0 do z := 0 od {x = 1}
(d) <x = 0> while z = 0 do z := 0 od <x = 0>
(e) {p} x := e {p $\wedge$ x = e}

E.III.15 Probar en H la fórmula de correctitud {p} while p do S od {q}, asumiendo que se puede probar en H la fórmula de correctitud {p $\vee$ q} S {p $\vee$ q}.

E.III.16 Verificar en H la correctitud parcial del siguiente programa, con respecto a la especificación planteada:

{x = X $\wedge$ y = Y $\wedge$ X < Y}
while x < y do x := x + 1 ; y := y – 1 od
{[par(X + Y) $\rightarrow$ x = y = (X + Y)/2] $\wedge$
[impar(X + Y) $\rightarrow$ x = (X + Y + 1)/2 $\wedge$ y = (X + Y – 1)/2]}

E.III.17 Formular una regla de prueba alternativa para la terminación de la instrucción while del lenguaje PLW en el método H*, en que se permitan decrementos arbitrarios del parámetro n, y no se exija que n alcance el valor cero a la terminación del while. Comentar la sensatez de la regla propuesta.

E.III.18 Probar en H* la correctitud total de los siguientes programas, con respecto a las especificaciones planteadas:

$<\exists k: k > 1 \wedge y = 3k>$
x := 0 ; while x < y do x := x + 1 ; y := y – 2 od
<y = x>

$<(\exists z \geq 0 : x = X = 2^z) \wedge (y = 0)>$
while par(x) do x := x / 2 ; y := y + 1 od
$<X = 2^y>$

E.III.19 Considérese la siguiente definición alternativa de la semántica del lenguaje PLW: por cada programa P y estado σ se define una secuencia de estados comp(P, σ); la secuencia de estados $<\sigma, \sigma_1, \sigma_2, \ldots, \sigma'> \models p$ si y sólo si $\sigma' \models p$. Si el símbolo «**.**» denota la concatenación de secuencias, se define formalmente:

a) $comp(skip)(\sigma) = \sigma$
b) $comp(x := e)(\sigma) = \sigma[x \mid e]$
c) $comp(S_1 ; S_2)(\sigma) = comp(S_1)(\sigma)$ **.** $comp(S_2)(\sigma')$, siendo σ' = $last(comp(S_1)(\sigma))$, es decir el último estado de la secuencia $comp(S_1)(\sigma)$
d) comp(if B then S_1 else S_2 fi)(σ) = $comp(S_1)(\sigma)$ si $\sigma \models B$, $comp(S_2)(\sigma)$ si $\sigma \models \neg B$

Se pide completar esta definición alternativa de la semántica de PLW (instrucción while), y probar la sensatez del método H con respecto a la misma.

E.III.20 La introducción de la estructura de datos de los arreglos en el lenguaje PLW impacta en la sensatez del método de prueba H, de acuerdo a cómo se definió dicho método y la semántica de PLW. Por ejemplo, se cumple $\vdash$ {true} a[a[2]]:=1 {a[a[2]]=1} por el axioma de asignación, pero no se cumple $\models$ {true} a[a[2]]:=1 {a[a[2]]=1}: si $\sigma \models p = \{a[1]=2 \wedge a[2]=2\}$, vale $\models$ {p} a[a[2]]:=1 {a[a[2]]=2 $\neq$1}.

Para preservar la sensatez de H, además de adecuar la definición de la expresión $p^{a[e]}_{e'}$ (esto no se va a desarrollar), siendo a un arreglo unidimensional (para simplificar la presentación se asumirá una sola dimensión), y e y e' expresiones enteras, se debe extender la definición de la semántica de PLW para considerar los arreglos. Se presenta a

continuación una posible extensión, basada en asignaciones a componentes de arreglos (en contraposición a la aproximación de asignaciones totales).

Primero, en lo que hace a la sintaxis, se define al conjunto de las variables enteras Ivar, con elementos v, w, …, como la unión disjunta del conjunto de las variables simples Svar, con elementos x, y, …, y el conjunto de las variables de tipo arreglo Avar, con elementos a[e], b[e'], … Y se mantienen los conjuntos Icon (constantes enteras m, n, …), Iexp (expresiones enteras e_1, e_2, …) y Bexp (expresiones booleanas B_1, B_2, …).

En el plano semántico se introduce el conjunto Intv = Svar $\cup$ (Avar x V), con elementos ε_1, ε_2, …, y ahora el conjunto de estados es $\Sigma = (\text{Intv} \to \text{V})$. La idea, que se formaliza después, es que para determinar el valor de una variable a[e], primero se asocia a a[e] el par <a, α>, siendo α la valuación de e en el estado corriente σ, y luego se aplica σ sobre <a, α>, obteniéndose α' como el valor de a[e]. Por ejemplo, dada la asignación a[e] := a[e'] y el estado σ: (a) se evalúa e dado σ, obteniéndose α, y se asocia a a[e] el par <a, α>; (b) se evalúa e' dado σ, obteniéndose α', y se asocia a a[e'] el par <a, α'>; (c) se aplica σ sobre el par <a, α'>, obteniéndose α'' para a[e']; y finalmente (d) se reemplaza σ por σ', que es igual a σ salvo que σ'(<a, α>) = α''. Es decir, es diferente el trato de una variable con subíndice en las partes izquierda y derecha de una asignación. En el paso anterior (a) se obtiene el «valor izquierdo» de a[e] dado σ, denotado con L(a[e])(σ), y en los pasos (b) y (c) en conjunto se obtiene el «valor derecho» de a[e'] dado σ, denotado con R(a[e'])(σ).

Para que todas las funciones semánticas sean totales, se mantiene el uso de elementos indefinidos en los distintos dominios: $\perp_{\text{Intv}}$ en Intv, $\perp_{\text{V}}$ en V, $\perp_{\text{W}}$ en W y $\perp$ en Σ. Se define:

1) $\varepsilon_1 = \varepsilon_2$ sii los dos son $\perp_{\text{Intv}}$, ó $\varepsilon_1 = \varepsilon_2 = x$ de Svar, ó $\varepsilon_1 = \langle a_1, \alpha_1 \rangle$ y $\varepsilon_2 = \langle a_2, \alpha_2 \rangle$ con $a_1 = a_2$ y $\alpha_1 = \alpha_2$
2) $\perp[\varepsilon|\alpha] = \perp$, y $\sigma[\varepsilon|\alpha](\varepsilon') = \alpha$ si $\varepsilon = \varepsilon'$ o bien $\sigma(\varepsilon')$ en caso contrario
3) $L : \text{Ivar} \to (\Sigma \to \text{Intv})$, con $L(v)(\perp) = \perp_{\text{Intv}}$, L(x)($\sigma$) = x, L(a[e])($\sigma$) = <a, R(e)($\sigma$)>
4) $R : \text{Iexp} \to (\Sigma \to \text{V})$, con $R(e)(\perp) = \perp_{\text{V}}$, R(m)($\sigma$) = α siendo α el entero representado por m, R(x)(σ) = σ(L(x)(σ)), $R(e_1 +$

$e_2)(\sigma) = R(e_1)(\sigma) + R(e_2)(\sigma)$ - de modo similar se definen las otras operaciones aritméticas -, R(if B then e_1 else e_2 fi)(σ) = if $\mathcal{W}$(B)(σ) then $R(e_1)(\sigma)$ else $R(e_2)(\sigma)$ fi

5) $\mathcal{W}$: Bexp $\to (\Sigma \to W)$ se define como antes, salvo $\mathcal{W}(e_1 = e_2)(\sigma)$ - y lo mismo considerando los operadores <, ≤, etc. - (queda como ejercicio este caso)
6) $\mathcal{M}$: PLW $\to (\Sigma \to \Sigma)$ se define como antes, salvo $\mathcal{M}$(v := e)(σ) (queda como ejercicio este caso)

Se pide, además de completar las definiciones previas (5) y (6), probar:

a) $\mathcal{M}$(x := x + 1)(σ[x|0]) = σ[x|1]
b) $\mathcal{M}$(a[a[2]] := 1)(σ[<a, 1> | 2][<a, 2> | 2]) = σ[<a, 1> | 2][<a, 2> | 1]

E.III.21 Proponer una transformación sintáctica en la invocación a un procedimiento con parámetros y en la declaración del mismo, para que pueda aplicarse después, sensatamente, la regla PAR del método H extendido para procedimientos, sin la restricción de que los parámetros reales pasados por valor-resultado sean dos a dos disjuntos (es decir, para que se puedan utilizar alias). Con la transformación sintáctica propuesta, probar:

```
{a > 0}
P :: procedure incdec (var x, i) :
        while i > 0 do x := x + 1 ; i := i – 1 od
     end ;
     call incdec (a, a)
{false}
```

E.III.22 Probar la correctitud parcial de los siguientes programas de PROC:

```
{a = N ≥ 0}
procedure incdec (val i ; var x):
while i > 0 do x := x+1; i := i – 1 od
end;
call incdec (a, a)
{a = 2N}
```

{n = N $\wedge$ s = 0}
procedure p:
if n $\leq$ 0 then skip
else n := n – 1; call p; s := s+1; call p; n := n+1 fi
end;
call p
{n = N s = 2^N – 1}

E.III.23 Probar la correctitud total del siguiente programa de PROC:

<x = N $\wedge$ N $\geq$ 0>
procedure d :
if x = 0 then skip else x := x – 1 ; call d ; x := x + 2 fi
end;
call d
< x = 2N >

E.III.24 Definir un sublenguaje determinístico del lenguaje GCL, equivalente al lenguaje PLW.

E.III.25 Indicar para qué casos de aserciones p y q de Assn, y de programas S de GCL, se cumple |= {p} S; abort {q}, siendo abort el programa [false $\rightarrow$ skip].

E.III.26 Indicar si se cumple (|= {p} S; loop {q}) $\leftrightarrow$ (|= {p} S {q}), siendo loop el programa del lenguaje GCL de la forma *[true $\rightarrow$ skip].

E.III.27 Indicar en qué circunstancias se cumple la fórmula de correctitud {p} x:=? {p}, siendo x := ? la asignación aleatoria de GCL.

E.III.28 Formular una regla en el método D para probar la «terminación existencial» de los programas de GCL, es decir, que dado un estado inicial σ que satisfaga una precondición p, exista al menos una computación que termine a partir de σ. Probar la sensatez de la regla propuesta.

E.III.29 Dados los siguientes programas de GCL:

(a) S_0 :: skip,

(b) S_1 :: *[true $\rightarrow$ skip],

(c) S_2 :: [true $\rightarrow$ skip $\square$ true $\rightarrow$ S],

indicar por pares si son equivalentes desde el punto de vista de la correctitud parcial, la correctitud total, y la terminación existencial.

E.III.30 Sean los programas P :: *[g → S] y P' :: *[g → S □ g → S]. Probar que se cumple:

$$\vdash_D \{p\}\ P\ \{q\} \leftrightarrow \vdash_D \{p\}\ P'\ \{q\}.$$

E.III.31 Probar en D* la correctitud total del siguiente programa de GCL:

$<(x_1, x_2, x_3, x_4) = (X_1, X_2, X_3, X_4)>$
$*[x_1 > x_2 \rightarrow (x_1, x_2) := (x_2, x_1)$
$\square$
$x_2 > x_3 \rightarrow (x_2, x_3) := (x_3, x_2)$
$\square$
$x_3 > x_4 \rightarrow (x_3, x_4) := (x_4, x_3)]$
$<(x_1, x_2, x_3, x_4) = \text{permutación}(X_1, X_2, X_3, X_4) \wedge x_1 \leq x_2 \leq x_3 \leq x_4>,$

donde $(x_a, x_b) := (x_b, x_a)$ es una asignación paralela que permite el intercambio de contenidos entre las variables x_a y x_b, y permutación(X_1, X_2, X_3, X_4) es una permutación de las variables X_1, X_2, X_3, X_4.

E.III.32 ¿Cuáles de los siguientes programas de GCL terminan sin hipótesis de fairness, cuáles con hipótesis de fairness débil, y cuáles con hipótesis de fairness fuerte, asumiendo la precondición true?

$*[x=0 \rightarrow y:=y+1 \square x=0 \rightarrow x:=1 \square x>0 \wedge y>0 \rightarrow y:=y-1]$

$*[b \rightarrow i:=i+1 \square b \rightarrow i:=0 \square b \wedge i=1 \rightarrow b:=\text{false}]$

$*[b \rightarrow i:=i+1 \square b \rightarrow i:=0 \square b \wedge i=2 \rightarrow b:=\text{false}]$

E.III.33 Probar la terminación del siguiente programa de GCL, asumiendo la existencia de fairness y la precondición true:

$*[xup \rightarrow x := x + 1$
$\square$
$yup \rightarrow y := y + 1$
$\square$
$xup \rightarrow xup := \text{false}$
$\square$
$yup \rightarrow yup := \text{false}$
$\square$
$x > 0 \rightarrow x := x - 1$
$\square$
$y > 0 \rightarrow y := y - 1]$

E.III.34 Suponiendo que se extiende el lenguaje PLW con la instrucción $[S_1 \,||\, S_2]$, cuya semántica informal es: «se ejecuta S_1 y luego se ejecuta S_2, o bien se ejecuta S_2 y luego se ejecuta S_1», definir formalmente la semántica operacional de dicha instrucción.

E.III.35 Probar la sensatez de la siguiente variante de la regla O-CONC, en el marco de la prueba de correctitud parcial de los programas SVL, tal que change(P_k) es el conjunto de variables que el proceso P_k modifica:

(1) $\{p_i\}\ S_i^*\ \{q_i\}$ proof outline de $\{p_i\}\ P_i\ \{q_i\}$, $i = 1,\ldots,n$

(2) $var(P_i) \cap change(P_k) = \varnothing$, con $i \neq k$

$\{p_1 \wedge \ldots \wedge p_n\}\ [P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n]\ \{q_1 \wedge \ldots \wedge q_n\}$

E.III.36 Indicar si puede probarse en el método O, sin utilizar la regla AUX, la fórmula de correctitud siguiente:

$\{true\}\ [x := 0 \,||\, await\ true \rightarrow x := x + 1\ end]\ \{x = 0 \vee x = 1\}$

E.III.37 Probar en el método O la correctitud parcial de los siguientes programas de SVL, con respecto a las especificaciones planteadas:

$\{true\}$
$[x := 1 ; x := 2 \,||\, await\ true \rightarrow x := x+1\ end]$
$\{x=2 \vee x=3\}$

$\{true\}$
$[await\ true \rightarrow x := x+1\ end ; await\ true \rightarrow x := x+1\ end \,||\, x := 0]$
$\{x=0 \vee x=1 \vee x=2\}$

E.III.38 Las operaciones p y v de un semáforo s podrían definirse en SVL de la siguiente manera:

p(s) :: await $s > 0 \rightarrow s := s - 1$ end
v(s) :: await $true \rightarrow s := s + 1$ end

Considérese la siguiente solución al problema de las secciones críticas (exclusión mutua):

SC :: s := 1; [P_1 || P_2], con:
P_i :: while true do
<sección no crítica i>;
p(s);
<sección crítica i>;
v(s)
od

Probar la ausencia de deadlock en el programa SC de SVL.

E.III.39 Sea el siguiente programa de SVL:

c := true ; p := false ;

[while true do		while true do
await c $\rightarrow$ c := false end ;		await p $\rightarrow$ p := false end ;
A_1 ;		A_2 ;
await $\neg$ p $\rightarrow$ p := true end ;	||	await $\neg$ c $\rightarrow$ c := true end ;
await c $\rightarrow$ c := false end ;		await p $\rightarrow$ p := false end ;
B_1 ;		B_2 ;
await $\neg$ p $\rightarrow$ p := true end ;		await $\neg$ c $\rightarrow$ c := true end ;
od		od]

Las secciones A1, A2, B1 y B2, no modifican las variables booleanas c ni p. Probar que los dos procesos nunca están simultáneamente dentro de sus secciones críticas A y B, es decir, que nunca los controles de los dos procesos están simultáneamente en A1 y A2, A1 y B2, B1 y A2, y B1 y B2.

E.III.40 Probar que en la configuración final de una computación que termina de un programa P de RVL con m recursos, se cumple para todo i, $1 \leq i \leq m$, que $\rho[i] = 0$.

E.III.41 Demostrar la sensatez de la regla WITH del método R para probar la correctitud parcial de los programas de RVL. Responder además: ¿por qué en la conclusión de la regla no se agrega, en la pre y postcondición, el invariante I_r asociado al recurso r?

E.III.42 Considérese la siguiente solución al mismo problema de exclusión mutua planteado en un ejercicio anterior en el marco del lenguaje SVL, ahora considerando el lenguaje RVL:

SC :: resource semáforo(s); s := 1;

[with semáforo when s = 1		with semáforo when s = 1
do s := 0		do s:=0
endwith;		endwith;
<sección crítica 1>;	||	<sección crítica 2>;
with semáforo when true		with semáforo when true
do s := 1		do s := 1
endwith		endwith]

Probar la ausencia de deadlock en el programa SC de RVL.

E.III.43 Supóngase que se tiene una proof outline concurrente del programa P_{fcp} de RVL presentado en el libro, relacionado con el problema de los filósofos que comen y piensan, tal que:

a) Las precondiciones de los with de los DP_i son p_1 = {eating[i] = 0} y p_2 = {eating[i] = 1}, respectivamente. El arreglo eating se agrega al programa original para indicar qué filósofos están comiendo en cada momento.

b) El invariante asociado al recurso forks se define de la siguiente manera:

$I_{forks} = \bigwedge_{i = 0...4}$ [(0 < eating[i] < 1) $\wedge$ (eating[i] = 1 $\rightarrow$ f[i] = 2) (f[i] = 2 – eating[i – 1] – eating[i + 1])].

Se pide probar que en el programa P_{fcp} no hay deadlock.

E.III.44 Supóngase que el lenguaje RVL provee monitores. De esta manera, ahora un recurso r tiene además de variables de programa, variables de condición $r.c_1$, $r.c_2$, …, $r.c_k$, asociadas a colas implícitas.

Existen dos operaciones que pueden realizarse, dentro del cuerpo de una sección crítica condicional asociada a un recurso r, sobre una variable de condición r.c. Una operación es wait r.c, con la que el proceso que la ejecuta queda suspendido y el recurso r se libera. La segunda operación es signal r.c, con la que algún proceso suspendido por un wait r.c retoma el control

sobre r y reanuda la ejecución del cuerpo de la sección crítica condicional inmediatamente después del wait (si hay más de un proceso, la selección es no determinística). El proceso que ejecuta el signal queda a su vez suspendido, y reanuda su ejecución dentro del cuerpo de la sección crítica condicional una vez que el recurso está libre otra vez. Una instrucción with relacionada con un recurso r puede llevarse a cabo sólo si no hay procesos suspendidos por operaciones signal asociados a r.

La extensión de la semántica operacional de RVL para contemplar el uso de monitores es la siguiente: a los arreglos ρ asociados a los m recursos se les agregan otros: γ_1, γ_2, …, γ_m, que contienen conjuntos de índices de procesos. El significado de $\gamma_k[h] = A$ es que para todo i ε A se cumple que P_i es un proceso suspendido por un wait $r_k.c_h$. Además se agrega el arreglo γ_0 [1…m], cada uno de cuyos elementos representa un conjunto de procesos que han llevado a cabo operaciones signal sobre alguna variable de condición del respectivo recurso.

Se pide describir formalmente la extensión de la semántica, e indicar además los valores de ρ y los distintos γ en la configuración inicial de un programa.

E.III.45 Dados el lenguaje CSP, un proceso P_i y un subprograma S de P_i, se define after (S,P_i) a la continuación sintáctica de P_i inmediatamente después de la ejecución de S. Definir formalmente after (S,P_i).

E.III.46 Dar una definición formal de proof outline asociada a un proceso de CSP.

E.III.47 Probar utilizando el método AFR:

(a) {y = Y} P:: [P_1:: P_2! y || P_2:: P_1? x] {x = Y}

(b) {true} P_{max} :: [P_1 || P_2] {u = max(x,y)}, siendo:
P_1 :: P_2 ? z ; [x ≥ z → P_2 ! x ☐ x ≤ z → P_2 ! z],
P_2 :: P_1 ! y ; P_1 ? u,
max(x,y) el máximo entre x e y.

E.III.48 Sea el programa:

P :: [P_1 || P_2 || P_3] , siendo:
P_1 :: P_2 ! 1

P_2 :: [true $\rightarrow$ P_1 ? x $\square$ x < 0 $\rightarrow$ P_3 ! x]
P_3 :: skip

(a) Plantear una postcondición q tal que valga {true} P {q}. Justificar (informalmente).
(b) ¿Se cumple <true> P <q>? Justificar (informalmente).
(c) Si la respuesta en (b) es negativa, plantear una precondición p tal que se cumpla <p> P <q>. Justificar (informalmente).

E.III.49 Sea el programa:

P :: [P_1 || P_2] , siendo:
P_1 :: [P_2 ? x $\rightarrow$ skip $\square$ P_2 ! 0 $\rightarrow$ P_2 ? x ; x := x + 1]
P_2 :: [P_1 ! 2 $\rightarrow$ skip $\square$ P_1 ? z $\rightarrow$ P_1 ! 1]

(a) Plantear una postcondición q tal que valga {true} P {q}. Justificar (informalmente).
(b) ¿Se cumple <true> P <q>? Justificar (informalmente).
(c) Plantear (sintácticamente) todos los casos posibles de deadlock.

E.III.50 Indicar si los siguientes programas de CSP terminan, sin y con hipótesis de fairness (débil y fuerte):

P::[P_1 || P_2], con:
P_1 :: b:=true ; *[b ; P_2? b $\rightarrow$ skip]
P_2 :: c := true ; *[c ; P_1! true $\rightarrow$ skip $\square$ c ; P_1! false $\rightarrow$ c := false]

P :: [P_1 || P_2 || P_3], con:
P_1 :: b := true ; *[b ; P_2? b $\rightarrow$ skip]
P_2 ::go:=true ; c:=true ; *[c ; P_1! go $\rightarrow$ [go $\rightarrow$ skip $\square$ $\neg$go $\rightarrow$ c:=false] $\square$ c ; P_3? go $\rightarrow$ skip]
P_3 :: P_2! false

P :: [P_1 || P_2 || P_3], con:
P_1 :: a_2 := true ; a_3 := true ; *[a_2 ; P_2? a_2 $\rightarrow$ skip $\square$ a_3 ; P_3? a_3 $\rightarrow$ skip]
P_2 :: b := true ; go_2 := true ; x := 0 ;
*[b ; P_1! go_2 $\rightarrow$ [$\neg go_2$ $\rightarrow$ b := false $\square$ go_2 $\rightarrow$ skip] $\square$ b ; P_3!x $\rightarrow$ go_2 := false]
P_3 :: c := true ; go_3 := true ; y := 0 ;
*[c ; P_1! go_3 $\rightarrow$ [$\neg go_3$ $\rightarrow$ c := false $\square$ go_3 $\rightarrow$ skip] $\square$ c ; P_2? y $\rightarrow$ go_3 := false]

E.III.51 Se indicó previamente en el libro que se cumple:

$$
\begin{array}{l}
\quad\quad <0 \le x \le 10 \wedge y \neq 0> \\
P :: [P_1 :: *[x \ge 0 \; ; \; P_2 \; ! \; x \rightarrow x := x - 1] \\
\quad\quad \| \\
\quad\quad P_2 :: *[y \neq 0 \; ; \; P_1 \; ? \; y \rightarrow skip]] \\
\quad\quad\quad\quad <true>
\end{array}
$$

Se pide probar formalmente la convergencia del programa P, dada la precondición planteada. Tener en cuenta que si bien tanto la iteración de P_1 como la de P_2 se efectúan x+1 veces, en P_2 el valor de y se conoce recién en medio de la primera repetición, por lo que será necesario utilizar una variable auxiliar z, por ejemplo con valor 0 antes de la primera repetición y con valor 1 después de la primera repetición, para distinguir las dos instancias. Así, quedaría:

$$
\begin{array}{l}
P' :: [P'_1 :: *[x \ge 0 \; ; \; P'_2 \; ! \; x \rightarrow x := x - 1] \\
\quad\quad \| \\
\quad\quad P'_2 :: z := 0 \; ; \; *[y \neq 0 \; ; \; P'_1 \; ? \; y \rightarrow skip \; ; \; z := 1]]
\end{array}
$$

Además, en el testeo de cooperación surgirá un problema de incompletitud: se deberá expresar la variable y en términos de x (es decir, y = x+1), y para ello tendrá que introducirse un invariante global.

E.III.52 Se pide definir la semántica operacional del lenguaje ACSP. Se considera, como en el caso de RVL, una tercera componente en las configuraciones, ahora para almacenar los mensajes enviados aún no recibidos. Esta componente puede implementarse mediante un arreglo γ, tal que $\gamma[c]$ tiene la secuencia de todos los mensajes enviados al canal c aún no recibidos. Una configuración tiene la forma $C = <[\|S_i], \gamma, \sigma>$. Asumir que se pueden emplear las operaciones habituales sobre secuencias: *enqueue* para encolar, *dequeue* para desencolar, y *first* para referirse al primer elemento de una secuencia. Podría tomarse como base lo hecho en el libro para CSP.

E.III.53 Sea el siguiente programa $P :: [P_1 \| P_2]$, escrito en un lenguaje que pretende implementar el modelo MCC, con $\mathbb{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, $X = \{x\}$, $C = \{c_1, c_2\}$, y:

$P_1 :: a_0$: $[\tau_1$: true $\rightarrow$ x:=x+1; goto a_0 $\square$ τ_2: true $\rightarrow$ goto $a_1]$

P_2 :: b_0: [τ_3: true $\rightarrow$ x:=x+1; goto b_0 $\square$ τ_4: impar(x) $\rightarrow$ goto b_1]
$\theta = (x = 0 \wedge c_1 = a_0 \wedge c_2 = b_0)$

Definir condiciones de fairness para que:

(a) P_1 termine. Expresar esta propiedad en el lenguaje de lógica temporal L definido.
(b) P termine. Expresar esta propiedad en el lenguaje L.

E.III.54 Determinar qué relación (equivalencia, implicación en un sentido u otro, ninguna) existe entre cada uno de los siguientes pares de fórmulas del lenguaje L:

a) $p \mathbb{U} (q \wedge r)$ vs $(p \mathbb{U} q) \wedge (p \mathbb{U} r)$
b) $\mathbb{GF}(p \vee q)$ vs $(\mathbb{GF}p \vee \mathbb{GF}q)$
c) $(p \mathbb{W}_U \mathbb{G}q)$ vs $(\mathbb{FG}q \vee \mathbb{G}p)$, con $(p \mathbb{W}_U q) = ((p \mathbb{U} q) \vee \mathbb{G}p)$; $\mathbb{W}_U$ es el operador weak until
d) $\mathbb{F}q$ vs $\mathbb{F}(p \mathbb{S} q)$
e) $((p \rightarrow (\mathbb{P}p \wedge \mathbb{N}p)) \wedge \mathbb{F}p)$ vs $\mathbb{G}p$, con $\mathbb{P}p$ = first $\vee \mathbb{Y}p$, $\mathbb{N}p$ = last $\vee \mathbb{X}p$, first = $\neg\mathbb{Y}$ true, last = $\neg\mathbb{X}$ true

E.III.55 Probar las siguientes fórmulas, en el marco de la lógica temporal presentada en el método de verificación T (resolver utilizando los axiomas, reglas y teoremas definidos):

a) $\mathbb{G}(p \wedge q) \rightarrow (\mathbb{G}p \wedge \mathbb{G}q)$
b) $(\mathbb{F}p \vee \mathbb{F}q) \rightarrow \mathbb{F}(p \vee q)$
c) $\mathbb{FF}p \rightarrow \mathbb{F}p$
d) Si $\vdash_T p \rightarrow q$, entonces $\forall n \geq 1 : \vdash_T (\mathbb{FG})^n p \rightarrow (\mathbb{FG})^n q$

E.III.56 Comentar la sensatez de las siguientes reglas del método de verificación T, que tienen como conclusión la fórmula $\mathbb{G}(p \rightarrow \mathbb{F}q)$:

Regla R3.
P1. $\mathbb{G}(p \rightarrow (q \vee \vartheta))$
P2. $\{\vartheta\}\tau\{q\}$, para toda transición τ

Regla R4 (se asume fairness fuerte).
P1. $\mathbb{G}(p \rightarrow (q \vee \vartheta))$
P2. $\{\vartheta\}\tau\{q\}$, para toda transición τ de un conjunto de transiciones $T_1 \neq \varnothing$
P3. $\{\vartheta\}\tau'\{q \vee \vartheta\}$, para toda transición τ' no perteneciente a T_1
P4. $\mathbb{GF}(\vartheta \rightarrow H(T_1))$; $H(T_1)$ significa que hay alguna transición de T_1 habilitada

E.III.57 Determinar qué relación (equivalencia, implicación en un sentido u otro, ninguna) existe entre cada uno de los pares siguientes de fórmulas del lenguaje de lógica temporal CTL*:

a) $\mathbb{E}((p \wedge q)\ \mathbb{U}\ r)$ vs $\mathbb{E}(p\ \mathbb{U}\ r) \wedge \mathbb{E}(q\ \mathbb{U}\ r)$
b) $\mathbb{A}(\mathbb{F}p \rightarrow \mathbb{F}q)$ vs $\mathbb{A}\mathbb{F}p \rightarrow \mathbb{A}\mathbb{F}q$
c) $\mathbb{A}(\mathbb{F}p \rightarrow \mathbb{F}q)$ vs $\mathbb{A}\mathbb{G}(p \rightarrow \mathbb{A}\mathbb{F}q)$
d) $\mathbb{A}\mathbb{G}(p \rightarrow \mathbb{F}q)$ vs $\mathbb{A}\mathbb{G}(p \rightarrow \mathbb{A}\mathbb{F}q)$
e) $\mathbb{A}\mathbb{F}\mathbb{G}p$ vs $\mathbb{A}\mathbb{F}\mathbb{A}\mathbb{G}p$
f) $\mathbb{A}\mathbb{G}\mathbb{F}p$ vs $\mathbb{A}\mathbb{G}\mathbb{E}\mathbb{F}p$
g) $\mathbb{A}\mathbb{G}\mathbb{E}q$ vs $\mathbb{A}\mathbb{G}q$
h) $\mathbb{A}\mathbb{G}\mathbb{E}(p \wedge q)$ vs $(\mathbb{A}\mathbb{G}\mathbb{E}p \wedge \mathbb{A}\mathbb{G}\mathbb{E}q)$
i) $\mathbb{E}\mathbb{G}(p \rightarrow p\ \mathbb{U}\ q)$ vs $\mathbb{E}(p \rightarrow \mathbb{G}(p\ \mathbb{U}\ q))$

E.III.58 Se define el lenguaje de lógica temporal RCTL como una extensión de CTL. La asunción básica de la semántica asociada en el marco de una estructura M es que el tiempo siempre progresa, es decir que no puede haber computaciones infinitas en las que el tiempo, a partir de un momento dado, deja de progresar. El tiempo se mide en unidades.

M se define así: dado un conjunto de proposiciones atómicas AP, $M = <\Sigma, R_0, R_1, L>$, donde Σ es un conjunto de estados, $R_0 \subseteq \Sigma \times \Sigma$ es un conjunto de transiciones que consumen 0 unidades de tiempo, $R_1 \subseteq \Sigma \times \Sigma$ es un conjunto de transiciones que consumen 1 unidad de tiempo, y $L : \Sigma \rightarrow \mathcal{P}(AP)$ asigna a cada estado el conjunto de proposiciones atómicas que valen en él. Se cumple que $(R_0 \cup R_1)$ es una relación total, es decir que de cualquier estado sale una transición de R_0 ó de R_1.

Un camino en M a partir de un estado se define así: a un estado σ de Σ se le asocia un número natural t, siendo t el tiempo transcurrido desde el inicio del camino. Un camino se denota así: (σ_0, t_0), (σ_1, t_1), (σ_2, t_2), …, siendo σ_0 el estado inicial del camino, t_0 el tiempo en σ_0, y además, para todo i: si $(\sigma_i, \sigma_{i+1}) \in R_0$, vale $t_i = t_{i+1}$, y si $(\sigma_i, \sigma_{i+1}) \in R_1$, vale $t_i + 1 = t_{i+1}$. Como el tiempo siempre progresa, se cumple que para todo número natural existe un índice i tal que $t_i = n$.

El lenguaje RCTL se define así:

(1) Si $A \in AP$, entonces A es una fórmula de RCTL.
(2) Si f, g $\in$ RCTL, entonces $\neg f$, $f \wedge g$, son también fórmulas de RCTL.

(3) Si f, g ε RCTL, entonces E[$X_{=k}$ f], E[f $\mathcal{U}_{\leq k}$ g], E[f $\mathcal{U}_{\geq k}$ g], para todo k ε N, son también fórmulas de RCTL.
(4) Como (3) pero cambiando E por A.
(5) Las únicas fórmulas de RCTL son las obtenidas de (1) a (4).

La semántica de RCTL se define así: dados M = <Σ, R_0, R_1, L>, (σ, t) y f, se define inductivamente:

(1) M, (σ, t) |= A $\leftrightarrow$ A ε L(σ)
(2) M, (σ, t) |= $\neg$ f $\leftrightarrow$ M, (σ, t) |$\neq$ f
(3) M, (σ, t) |= f $\wedge$ g $\leftrightarrow$ M, (σ, t) |= f $\wedge$ M (σ, t) |= g
(4) M, (σ, t) |= E[$X_{=k}$ f] $\leftrightarrow$ hay un camino en M desde (σ, t) y existe i $\geq$ 0 tal que se cumple (σ_i, t_i) |= f, t_i = t + k, t_{i-1} < t + k. Asumir de ahora en más (σ, t) = (σ_0, t_0). En palabras, (σ_i, t_i) es el primer estado del camino en el que el tiempo es (t + k) y vale f.
(5) M, (σ, t) |= E[f $\mathcal{U}_{\leq k}$ g] $\leftrightarrow$ hay un camino en M desde (σ, t) y existe i $\geq$ 0 tal que se cumple (σ_i, t_i) |= g, $t_i \leq$ t + k, y para todo 0 $\leq$ j < i, (σ_j, t_j) |= f. En palabras, se cumple (f $\mathcal{U}$ g) antes que el tiempo exceda las k unidades desde (σ, t).
(6) M, (σ, t) |= E[f $\mathcal{U}_{\geq k}$ g] $\leftrightarrow$ hay un camino en M desde (σ, t) y existe i $\geq$ 0 tal que se cumple (σ_i, t_i) |= g, $t_i \geq$ t + k, y para todo 0 $\leq$ j < i, (σ_j, t_j) |= f $\wedge$ $\neg$ g. En palabras, se cumple (f $\mathcal{U}$ g) desde (σ, t) por primera vez a las k o más unidades de tiempo.
(7) Como (4) a (6) pero cambiando E (hay un camino en M tal que …) por A (en todos los caminos de M se cumple que…).

Se pide:

a) Definir la semántica de la fórmula E[f $\mathcal{U}_{=k}$ g], tal que exprese que (f $\mathcal{U}$ g) se cumple en un camino, a partir de (σ, t), por primera vez, exactamente cuando el tiempo pasa a valer t + k.
b) Definir los operadores $EF_{\leq k}$, $EF_{\geq k}$, $EG_{\leq k}$, $EG_{\geq k}$, en términos de los operadores definidos anteriormente. Explicar en palabras qué expresa cada uno.
c) Responder: ¿Se pueden expresar los operadores EX y E$\mathcal{U}$ de CTL en términos de los operadores de RCTL? Justificar.
d) Escribir una fórmula de RCTL que sea verdadera en (σ, t) sii existe un camino desde (σ, t) en el que todos los estados con tiempo (t + k) satisfacen f.

E.III.59 Dado el opc (orden parcial completo) $(D = (\Sigma \to \Sigma), \leq_2)$ utilizado para definir denotacionalmente la semántica de la instrucción while de PLW, se define que una función $\vartheta \in D$ es monótona sii para todo par σ_1 y σ_2 de Σ, $\sigma_1 \leq_1 \sigma_2 \to \vartheta(\sigma_1) \leq_1 \vartheta(\sigma_2)$. Probar, considerando ahora el opc $(D_e, \leq_2)$, en que D_e sólo incluye las funciones estrictas de D, que todas las funciones de D_e son monótonas.

E.III.60 Probar que:

a) $(\mathcal{T}, \leq_{\mathcal{T}})$, el orden de Egli-Milner utilizado para la definición denotacional de la semántica de GCL, es efectivamente un opc.

b) $(E_e = (\Sigma \to \mathcal{T}), \leq)$, tal como se definió en el libro, es efectivamente un opc.

E.III.61 Existen dos opc alternativos al de Egli-Milner. Dado el opc discreto $(\Sigma, \leq_1)$, se había definido:

- opc de Egli-Milner $(P^E, \leq_E)$, con:
  $P^E = \{C \mid C \in \mathcal{P}(\Sigma), C \text{ finito}, C \neq \emptyset\} \cup \{C \mid C \in \mathcal{P}(\Sigma), C \text{ infinito}, \perp \in C\}$, y
  $\forall A,B$ de P^E vale $A \leq_E B \leftrightarrow ((\perp \in A \wedge A - \{\perp\} \subseteq B) \vee (\perp \notin A \wedge A = B))$.

Ahora se definen además:

- opc de Hoare $(P^H, \leq_H)$, con:
  $P^H = \{C \mid C \in \mathcal{P}(\Sigma), \perp \in C\}$, y
  $\forall A,B$ de P^H vale $A \leq_H B \leftrightarrow (\forall \sigma_1 \in A, \exists \sigma_2 \in B: \sigma_1 \leq_1 \sigma_2)$.
- opc de Smyth $(P^S, \leq_S)$, con:
  $P^S = \{C \mid C \in \mathcal{P}(\Sigma), C \text{ finito}, C \neq \emptyset, \perp \notin C\} \cup \{\Sigma\}$, y
  $\forall A,B$ de P^S vale $A \leq_S B \leftrightarrow (\forall \sigma_2 \in B, \exists \sigma_1 \in A: \sigma_1 \leq_1 \sigma_2)$.

Responder:

(a) En el opc de Egli-Milner, informalmente la idea es que B está más definido que A si todos los estados definidos de A están en B y no puede ser que B tenga el estado indefinido y A no. Expresar informalmente qué significa que B está más definido que A en los opc de Hoare y Smyth.

(b) Asociar cada una de las aseveraciones siguientes a uno o más de los tres opc referidos: (b1) El pasaje de una

aproximación a otra mejor no reduce el grado de no determinismo (es decir, no reduce la cantidad de estados). (b2) El pasaje de una aproximación a otra mejor no aumenta el grado de no determinismo.

(c) Probar: $\forall$ A,B de P^H vale $A \leq_H B \leftrightarrow A \subseteq B$.

(d) Probar: $\forall$ A de P^S vale $\perp \varepsilon A \rightarrow (\forall B \text{ de } P^S : A \leq_S B)$.

(e) Probar: $\forall$ A,B de P^S vale $\perp \varepsilon (A \cap B) \rightarrow (A \leq_S B \wedge B \leq_S A)$.

(f) Probar: $\forall$ A,B de P^S vale $\perp \notin (A \cup B) \rightarrow (A \leq_S B \leftrightarrow B \subseteq A)$.

(g) Probar que el orden de Hoare es efectivamente un opc, y explicar por qué si P^H fuera $\mathcal{P}(\Sigma) - \{\varnothing\}$ dejaría de serlo.

(h) Probar que el orden de Smyth es efectivamente un opc, y explicar por qué si P^S tuviera además de Σ otro conjunto que incluya $\perp$ dejaría de serlo.

(i) Probar: $\forall$ A,B de P^E vale $A \leq_E B \leftrightarrow (A \leq_H B \wedge A \leq_S B)$.

(j) Sea S un programa de GCL y $\mathcal{M}(S)(\sigma) = A$, siendo A un conjunto del opc de Egli-Milner. Caracterizar la forma de A en cada uno de los siguientes casos: (j1) Al menos una computación de S es finita. (j2) Todas las computaciones de S terminan. (j3) Ninguna computación de S termina. (j4) Si S termina se cumple la condición q.

Los autores

Ricardo Rosenfeld (rrosenfeld@pragmaconsultores.com) obtuvo el título de Calculista Científico de la Facultad de Ciencias Exactas de la Universidad Nacional de La Plata, Argentina, en 1983, y completó los estudios de la Maestría en Ciencias de la Computación del Instituto de Tecnología Technión, Israel, en 1991.
Desde 1991 se desempeña como Profesor en la Universidad Nacional de La Plata, en las áreas de teoría de la computación y verificación de programas. Previamente, entre los años 1984 y 1990, fue Auxiliar Docente en la Universidad Nacional de La Plata (lenguajes y metodologías de programación), en la Universidad de Buenos Aires (verificación y derivación de programas), en la Escuela Superior Latinoamericana de Informática (algorítmica y estructuras de datos, y teoría de compiladores), y en el Instituto de Tecnología Technión de Israel (programación).
Es además uno de los socios de Pragma Consultores, empresa regional dedicada a la tecnología de la información, ingeniería de software y consultoría de negocios.

Jerónimo Irazábal (jirazabal@lifia.info.unlp.edu.ar) obtuvo el título de Licenciado en Informática de la Facultad de Informática de la Universidad Nacional de La Plata, Argentina, en 2009.
Desde 2005 se desempeña como Auxiliar Docente en la Universidad Nacional de La Plata, en las áreas de teoría de la computación y verificación de programas, organización de computadoras, y arquitectura de computadoras. Previamente, entre los años 2003 y 2005, fue becario del Laboratorio de Investigación y Formación en Informática Aplicada de la Universidad Nacional de La Plata, y miembro de la Comisión de Investigación de Ciencias y Tecnologías de la misma universidad.
Es además uno de los socios de Eureka Consulting, empresa dedicada al desarrollo de software.

Esta publicación se terminó de imprimir
en el mes de marzo de 2010,
en la ciudad de La Plata,
Buenos Aires,
Argentina.