COMPUTABILIDAD, COMPLEJIDAD COMPUTACIONAL Y VERIFICACIÓN DE PROGRAMAS

Ricardo Rosenfeld y Jerónimo Irazábal

Los autores

Ricardo Rosenfeld
Fac. de Informática, UNLP
rrosenfeld@pragmaconsultores.com

Jerónimo Irazábal
Fac. de Informática, UNLP
jirazabal@lifia.info.unlp.edu.ar

Ricardo Rosenfeld obtuvo en 1983 el título de Calculista Científico de la Facultad de Ciencias Exactas de la Universidad Nacional de La Plata (UNLP), Argentina, y en 1991 completó los estudios de la Maestría en Ciencias de la Computación del Instituto de Tecnología Technión, Israel. Desde 1991 se desempeña como Profesor en la UNLP, en las áreas de teoría de la computación y verificación de programas. Previamente, entre 1984 y 1990, fue docente en la UNLP (lenguajes y metodologías de programación), en la Universidad de Buenos Aires (verificación y derivación de programas), en la Escuela Superior Latinoamericana de Informática (algorítmica, estructuras de datos, 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. Con Jerónimo Irazábal publicó en 2010 el libro Teoría de la Computación y Verificación de Programas, editado por la EDULP en conjunto con McGraw-Hill.

Jerónimo Irazábal obtuvo en 2009 el título de Licenciado en Informática de la Facultad de Informática de la UNLP, y actualmente se encuentra desarrollando un Doctorado en Ciencias Informáticas en la misma Facultad, centrado en los lenguajes de dominios específicos. Desde 2005 es docente en la UNLP, en las áreas de teoría de la computación, verificación de programas y lógica. Además es becario del Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), y su lugar de trabajo es el Laboratorio de Investigación y Formación en Informática Avanzada (LIFIA). Anteriormente participó en la creación de la compañía Eureka Consulting S.A., dedicada al desarrollo de software. Es co-autor, con Ricardo Rosenfeld, del libro Teoría de la Computación y Verificación de Programas.

Índice general

Prólogo

Computabilidad, Complejidad Computacional y Verificación de Programas contiene las quince clases que conforman la asignatura Teoría de la Computación y Verificación de Programas, una introducción a la teoría de la computabilidad y complejidad computacional de problemas y la teoría de correctitud de programas, que dicto en la Licenciatura en Informática de la Facultad de Informática de la Universidad Nacional de La Plata desde hace varios años. El libro es una suerte de segunda edición reducida de *Teoría de la Computación y Verificación de Programas*, de los mismos autores, editado en 2010 por la EDULP conjuntamente con McGraw-Hill, el cual incluye además de las clases de la asignatura básica, las de Teoría de la Computación y Verificación de Programas Avanzada, asignatura que también dicto en la misma carrera desde hace tiempo. El nuevo trabajo excluye principalmente la complejidad espacial, la verificación de los programas no determinísticos y concurrentes, el empleo de la lógica temporal para verificar los programas reactivos, y la semántica denotacional de los lenguajes de programación, tópicos tratados en la obra anterior. De todos modos, en la presente publicación hay secciones, breves, dedicadas a la jerarquía espacial, la terminación con hipótesis de *fairnes* de los programas no determinísticos, y la verificación de los programas concurrentes con memoria compartida, desarrolladas de la manera en que dichos temas son referenciados en la asignatura básica.

A sólo tres años de la publicación anterior, este libro se justifica por varias razones:

- Se profundiza en determinados contenidos, por ejemplo en la jerarquía temporal y la sensatez y completitud de los métodos de verificación de programas, que por cuestiones de espacio, dada la abundancia de temas tratados, en el trabajo de 2010 no se pudieron presentar de una manera más extensa.
- Se modifica la exposición de algunos temas, por ejemplo la caracterización de los problemas de la clase NP y las definiciones iniciales de la verificación de programas, a partir de la lectura de nuevos trabajos o de cambios en la modalidad del dictado de las clases respectivas. Asimismo, y ahora con respecto a todo el material, la presentación de los temas es más discursiva que en el libro anterior.

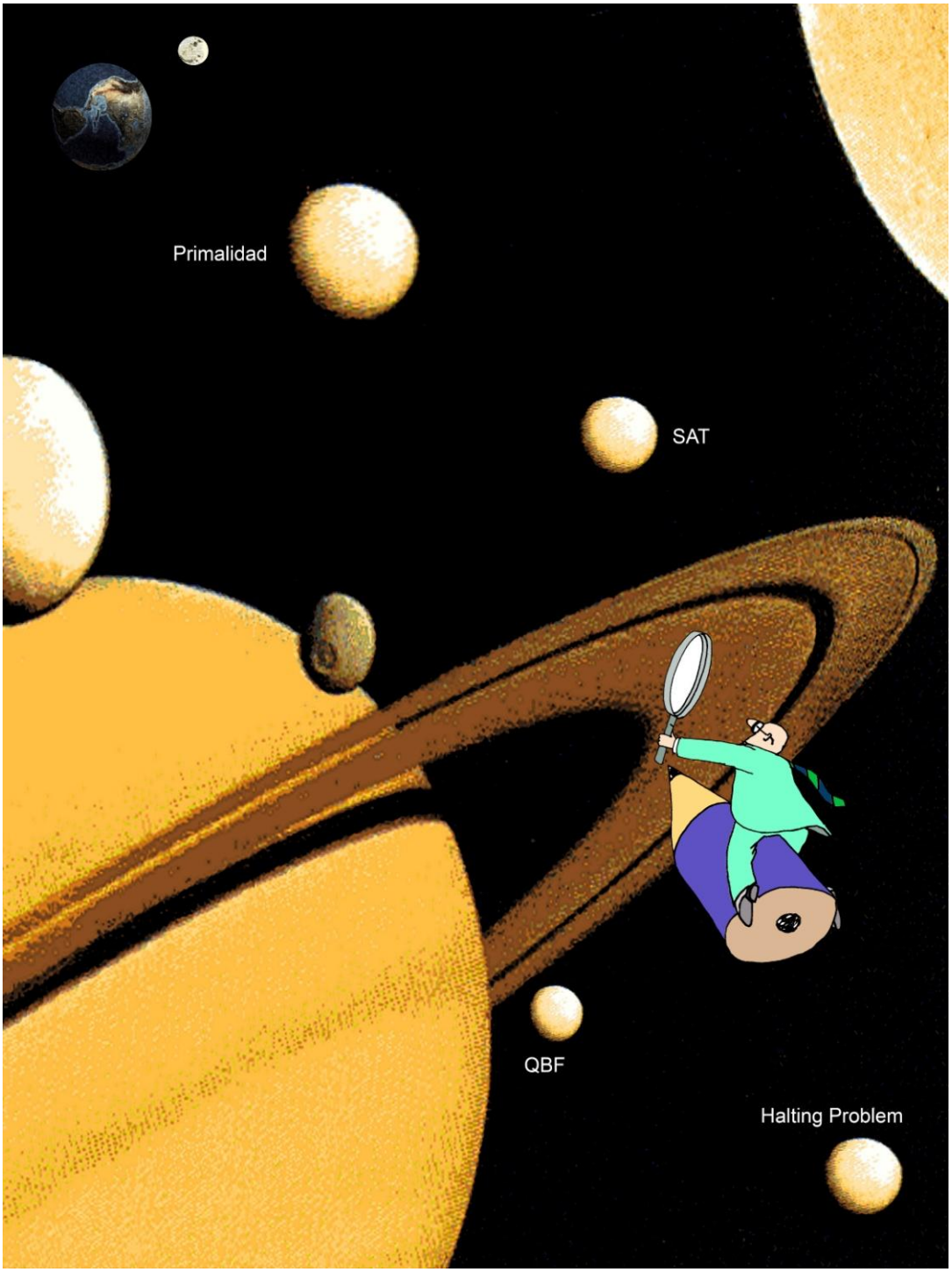
- Se desarrollan más ejemplos, más que nada los relacionados con las reducciones de problemas tanto en la parte dedicada a la computabilidad como en la parte dedicada a la complejidad computacional (en este último caso se trata de las reducciones polinomiales).
- Se adapta más fielmente la forma del libro a la del dictado de la asignatura asociada. La estructura general consiste en tres partes (computabilidad, complejidad computacional y verificación de programas). Cada parte está compuesta por cinco clases, y sus componentes básicos son definiciones, teoremas, ejemplos y ejercicios. La quinta clase de cada parte presenta misceláneas de la disciplina considerada.

Por lo demás, las características de este trabajo son muy similares a las del trabajo anterior. Se asume que los lectores ya han adquirido cierta madurez matemática, y sólidos conocimientos en algorítmica, estructuras de datos, y lenguajes y paradigmas de programación. Las quince clases están dirigidas principalmente a los estudiantes de los últimos años de las carreras de computación, a los graduados que desean introducirse en la teoría de la computación y la teoría de correctitud de programas, y naturalmente a los docentes que dictan o quieran dictar contenidos afines. En particular, a estos últimos les recomiendo no obviar ningún tema cuando desarrollen esta asignatura; si el material completo resulta excesivo, podrían prescindir de algunos ejemplos, priorizando las necesidades de los estudiantes.

El libro se puede tomar como un viaje imaginario a lo largo del universo de problemas. Se arranca en los confines de dicho universo y se viaja hacia su interior (algo así como se muestra en la figura de la página siguiente). En un primer tramo se atraviesan las fronteras de la computabilidad y la decidibilidad (primera parte del libro). En el tramo siguiente, ya en medio de los problemas computables decidibles, se lleva a cabo una exploración sobre los mismos con dos objetivos, el análisis de la complejidad temporal (segunda parte del libro) y la verificación de la correctitud de los algoritmos (o programas) construidos para resolverlos (tercera parte del libro).

En el desarrollo de los temas se trata de combinar rigor matemático con informalidad, apuntando a introducir los conceptos básicos de una manera precisa pero al mismo tiempo didáctica. El objetivo no es exponer un compendio de resultados y ejemplos, sino plantear fundamentos de las áreas estudiadas (cabe la comparación con la enseñanza de las matemáticas, parafraseando a un reconocido investigador: en lugar de

proveer recetas para derivar e integrar funciones, este libro estaría más cerca de explicar nociones básicas del tipo continuidad y límite de una función). En todos los casos se pretende emplear los componentes más intuitivos y difundidos en la literatura para



presentar los temas. Así, por ejemplo, se utilizan las máquinas de Turing como modelo de ejecución de los algoritmos, el tiempo como recurso computacional para medir la

complejidad de un problema, y la semántica operacional como técnica para especificar formalmente un lenguaje de programación.

Al final de cada una de las tres partes del libro se presentan referencias históricas y bibliográficas. La bibliografía recomendada se concentra en las obras más clásicas y abarcativas (de las que se puede obtener mayor detalle). Las definiciones, teoremas y ejemplos que se quieren destacar se presentan entre delimitadores bien marcados, como si fueran subsecciones, y al final del libro se les hace referencia mediante índices específicos. No siempre los teoremas se presentan en sus formas originales; a veces sus desarrollos se adaptan con fines didácticos. Para involucrar al lector en la terminación de la prueba de un teorema, o para reforzar su entendimiento acerca de una definición, además de los ejercicios que se plantean al final de una clase se intercalan otros en medio de la misma (que de todos modos se vuelven a enunciar al final). Se trata de utilizar la terminología más difundida de las distintas disciplinas, generalmente en castellano (hay algunas pocas excepciones en que se emplean términos en inglés por ser justamente más conocidos, como por ejemplo *deadlock* y *fairness*).

Agradezco en primer lugar a Jerónimo Irazábal, co-autor, además de colaborador en el dictado de las dos asignaturas que mencioné al comienzo; discutió conmigo la selección y redacción de todo el material del libro. Agradezco a las autoridades de la Facultad de Informática de la Universidad Nacional de La Plata, por la nueva posibilidad para publicar estos contenidos. Agradezco a Victoria Meléndez, quien realizó todas las ilustraciones. Agradezco a mi familia, por su aliento permanente para que lleve adelante esta iniciativa. Y como en la anterior oportunidad, agradezco especialmente a mis alumnos, fuente permanente de sugerencias de mejoras para la enseñanza de los tópicos tratados.

Deseo que este trabajo sirva como guía de estudio para los alumnos y guía de enseñanza para los docentes, que sea estímulo para profundizar en la teoría de la computación y la verificación de programas, y que las clases les resulten a los lectores tan fascinantes (¡y entretenidas!) como a los autores.

Ricardo Rosenfeld

La Plata, diciembre de 2012

Parte 1. Computabilidad

Las cinco clases de la primera parte del libro están dedicadas a la *computabilidad* de los problemas. Se define un modelo de ejecución específico, y se establece en términos del mismo la frontera de "lo computable" dentro del universo de los problemas. Se consideran particularmente los problemas computables decidibles, que se resuelven algorítmicamente a partir de todo dato de entrada (no es el caso general).

En la Clase 1 se define el modelo de ejecución, el más difundido en la literatura: las *máquinas de Turing*. Se presentan las tres visiones conocidas de las máquinas de Turing, como calculadoras de funciones, reconocedoras de lenguajes y generadoras de lenguajes. Como se tratan mayoritariamente los problemas de decisión (tienen por solución simplemente la respuesta "sí" o la respuesta "no"), las máquinas de Turing que reconocen lenguajes son las más utilizadas, dado que se puede establecer fácilmente una correspondencia entre las instancias positivas (negativas) de un problema de decisión y las cadenas pertenecientes (no pertenecientes) a un lenguaje. Luego de la introducción de un modelo inicial de las máquinas de Turing se describen otros modelos equivalentes, entre los que se destaca el de las máquinas de Turing con varias cintas y sólo dos estados finales (de aceptación y rechazo), seleccionado como modelo estándar para las definiciones, teoremas y ejemplos.

En la Clase 2 se describe la *jerarquía de la computabilidad*, es decir el "mapa" de las distintas clases de lenguajes (o problemas de decisión) en el que se distribuye, considerando la computabilidad, el universo de los lenguajes, generados a partir de un alfabeto universal Σ. Se presentan los lenguajes recursivamente numerables (clase RE), asociados a los problemas computables (por máquinas de Turing), y los lenguajes recursivos (clase R), asociados a los problemas decidibles dentro de los computables. Como consecuencia, quedan establecidos formalmente los límites de la computabilidad y la decidibilidad. Se describen propiedades de los lenguajes recursivamente numerables y recursivos, destacándose un teorema que caracteriza a la clase R como la intersección entre las clases RE y CO-RE, siendo esta última la que nuclea a los lenguajes cuyos complementos con respecto a Σ* están en RE.

El tema central de la Clase 3 es la *indecidibilidad*. Se describe la máquina de Turing universal, que en su forma habitual tiene al comienzo en su cinta de entrada el código de una máquina de Turing y un dato. De este modo se trabaja con un modelo de ejecución

más parecido al de las computadoras reales, con la noción de programa almacenado. Se define una manera de codificar las máquinas de Turing. Mediante la técnica de diagonalización se introduce un primer lenguaje de la clase RE – R, es decir un primer problema computable que no es decidible: el problema de la detención de las máquinas de Turing. Con la misma técnica se muestran lenguajes que ni siquiera son recursivamente numerables, es decir problemas por fuera de los límites de la computabilidad.

Prosiguiendo con la indecidibilidad como tema central, en la Clase 4 se describen las *reducciones de problemas*, técnica mediante la cual se pueden poblar de manera sistemática las distintas clases de la jerarquía de la computabilidad. Se desarrollan numerosos ejemplos de reducciones de problemas, entre los que se destaca la reducción que prueba la indecidibilidad de la lógica de primer orden (no se puede determinar en todos los casos si una fórmula es válida, o lo que es lo mismo, si es un teorema). A partir de una reducción de problemas particular que constituye la prueba de un teorema conocido como el Teorema de Rice, se plantea otra técnica con la que se puede demostrar fácilmente que un determinado tipo de problemas es indecidible.

Finalmente, en la Clase 5 se presentan distintas *miscélaneas de computabilidad*. Se introducen las máquinas RAM (máquinas de acceso aleatorio), más parecidas a las computadoras reales que las máquinas de Turing, y se prueba la equivalencia entre ambos modelos. Se prueba también la equivalencia entre las máquinas de Turing que reconocen y que generan lenguajes. Se describe una manera alternativa de representar lenguajes, mediante gramáticas. Se introducen distintos tipos de máquinas de Turing restringidas, con un poder computacional menor al de las máquinas de Turing generales de acuerdo a sus fines específicos, y como consecuencia queda establecida una categorización de los lenguajes asociados a dichas máquinas. Las máquinas de Turing restringidas presentadas son los autómatas finitos, los autómatas con pila y los autómatas acotados linealmente, y los lenguajes que reconocen son, respectivamente, los lenguajes regulares, los lenguajes libres de contexto y los lenguajes sensibles al contexto (incluyendo a los lenguajes recursivamente numerables sin ningún tipo de restricción, a esta clasificación de lenguajes se la conoce como jerarquía de Chomsky). La última miscelánea trata las máquinas de Turing con oráculo, útiles entre otras cosas para establecer relaciones entre los lenguajes de las distintas clases de la jerarquía de la computabilidad. Relacionadas con las máquinas de Turing con oráculo, se introducen las Turing-reducciones.

Clase 1. Máquinas de Turing

El objetivo de esta clase es introducir las *máquinas de Turing*. Como es habitual, utilizaremos dichas máquinas como modelo de ejecución de los algoritmos, para el estudio de la computabilidad en la primera parte de este libro y de la complejidad computacional temporal en la segunda parte. Las máquinas de Turing se presentaron por primera vez en 1936 en una publicación de A. Turing. Desde entonces, la conjetura conocida como Tesis de Church-Turing, de que todo lo computable puede ser llevado a cabo por una máquina de Turing, no ha sido refutada.

Una máquina de Turing es un artefacto que resuelve problemas (computacionales), en su visión más general produciendo o *calculando* una solución. Por ejemplo, dado el problema de encontrar un camino del vértice v_1 al vértice v_2 en un grafo, una máquina de Turing que lo resuelve produce, a partir de una instancia, en este caso un grafo G, un camino en G del vértice v_1 al vértice v_2, si es que existe.

Una segunda visión de máquina de Turing más restringida, que utilizaremos la mayoría de las veces para simplificar las exposiciones sin perder en esencia generalidad, como comprobaremos más adelante, es la de una máquina que sólo puede resolver un *problema de decisión*, produciendo únicamente la respuesta sí o no ante una instancia. Volviendo al problema ejemplo, con esta visión una máquina que lo resuelve acepta los grafos que tienen un camino del vértice v_1 al vértice v_2, y rechaza al resto. Esta es la visión de máquina de Turing *reconocedora*. Se denomina así porque la resolución de un problema consiste en reconocer el lenguaje que representa el problema, más precisamente, el lenguaje de las cadenas de símbolos que representan las instancias positivas del problema (en el ejemplo, las cadenas que representan grafos con un camino del vértice v_1 al vértice v_2).

Hay todavía una tercera visión de máquina de Turing, según la cual se resuelve un problema *generando* todas sus instancias positivas (en el caso del problema del camino en un grafo, generando todos los grafos con un camino del vértice v_1 al vértice v_2). Demostraremos después la equivalencia entre las máquinas generadoras y reconocedoras.

Vamos a trabajar con las tres visiones de máquina de Turing, calculadora, reconocedora y generadora. Ya dijimos que en la gran mayoría de los casos consideraremos la máquina reconocedora. Más aún, salvo mención en contrario, los problemas deberán

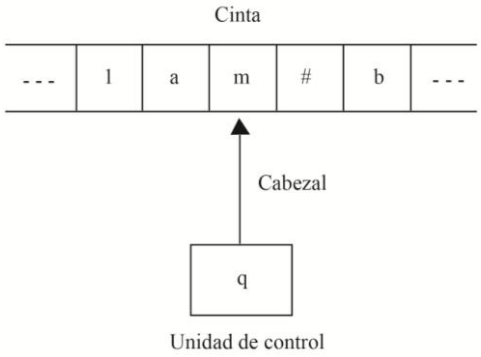
entenderse como problemas de decisión, y así los términos problema y lenguaje se tratarán de manera indistinta. Al final de la segunda parte del libro se verán con cierta profundidad los problemas más generales, conocidos como *problemas de búsqueda*.
A continuación introducimos formalmente las máquinas de Turing, y presentamos algunos ejemplos. Luego definimos distintos modelos de máquinas de Turing y probamos que son equivalentes.

Definición 1.1. Máquina de Turing

Una *máquina de Turing* M (de ahora en más utilizaremos la abreviatura 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 siguiente muestra los componentes de una máquina de Turing:



Los estados pertenecen a un conjunto Q, y los símbolos a un alfabeto Γ. Al comienzo, en la *configuración* o *instancia inicial* de M, la cinta tiene la *cadena de entrada* (o simplemente la *entrada*), limitada a izquierda y derecha por infinitos símbolos blancos, que se denotan con B. La unidad de control almacena el *estado inicial*, denotado en general con q_0. Y el cabezal apunta al primer símbolo de la entrada, que es su símbolo

de más a la izquierda. Si la entrada es la cadena vacía, denotada con λ, entonces el cabezal apunta a algún blanco.

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, es decir si M no se detiene, se dice que M tiene o entra en un *loop* (bucle).

Formalmente, una máquina de Turing M es una 6-tupla $(Q, \Sigma, \Gamma, \delta, q_0, F)$, tal que:

- Q es el conjunto de estados de M.
- Σ es el alfabeto de las entradas de M.
- Γ es el alfabeto de las cadenas de la cinta de M. Por convención, $B \in (\Gamma - \Sigma)$.
- δ es la función de transición de M. Se define $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$, 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.
- F es el conjunto de estados finales de M (su significado se aclara a continuación).

Considerando la visión de MT reconocedora de un lenguaje, si a partir de la entrada 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 las cadenas aceptadas por la MT M es el lenguaje aceptado o reconocido por M, y se denota con L(M). Considerando la visión de MT M calculadora, sólo cuando M se detiene en un estado $q \in F$ debe tenerse en cuenta el contenido final de la cinta, es decir la *cadena de salida* (o simplemente la *salida*).

Fin de Definición

Los siguientes son dos ejemplos de MT, uno con la visión de máquina reconocedora y el otro con la visión de máquina calculadora. Más adelante consideraremos las MT generadoras. Luego de los ejemplos trabajaremos en general solamente con la visión de MT reconocedora.

Ejemplo 1.1. Máquina de Turing reconocedora

Sea el lenguaje L = $\{a^n b^n \mid n \geq 1\}$. Es decir, L tiene infinitas cadenas de la forma ab, aabb, aaabbb, … Se va a construir una MT M que acepta L, o en otras palabras, tal que L(M) = L. En este caso, el lenguaje L representa directamente un problema de reconocimiento de las cadenas de un lenguaje.

Idea general.

Por cada símbolo *a* que lee, la MT M 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 del símbolo *a* que está inmediatamente a la derecha de la *a* anterior. Si al final del proceso no quedan símbolos por reemplazar, la MT M se detiene en un estado de F, porque significa que la entrada tiene la forma $a^n b^n$, con n ≥ 1. En caso contrario, M se detiene en un estado de (Q – F).

Construcción de la MT M.

La MT M = $(Q, \Sigma, \Gamma, \delta, q_0, F)$ es 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 símbolos *a*. El estado q_F es el estado final.
- $\Sigma = \{a, b\}$
- $\Gamma = \{a, b, \alpha, \beta, B\}$
- $q_0 = q_a$
- $F = \{q_F\}$
- 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. $\delta(q_b, \beta) = (q_b, \beta, R)$
  5. $\delta(q_L, \beta) = (q_L, \beta, L)$
  6. $\delta(q_L, a) = (q_L, a, L)$
  7. $\delta(q_L, \alpha) = (q_a, \alpha, R)$
  8. $\delta(q_a, \beta) = (q_H, \beta, R)$
  9. $\delta(q_H, \beta) = (q_H, \beta, R)$
  10. $\delta(q_H, B) = (q_F, B, S)$

Prueba de L(M) = L.

a. Si w ∈ L, entonces w tiene la forma $a^n b^n$, con n ≥ 1. Por cómo está definida la función de transición δ, claramente a partir de w la MT M acepta w, es decir que w ∈ L(M).
b. Si w ∉ L, entonces M rechaza w, es decir que w ∉ L(M). Por ejemplo, si w = λ, M rechaza porque no está definido en δ el par (q_a, B). Si w tiene un símbolo distinto de *a* o de b, M rechaza porque dicho símbolo no está considerado en δ. Si w empieza con b, M rechaza porque no está definido en δ el par (q_a, b). Queda como ejercicio completar la prueba.

Fin de Ejemplo

Ejemplo 1.2. Máquina de Turing calculadora

Se va a construir una MT M que calcula la resta m – n, tal que m y n son dos números naturales representados en notación unaria. Cuando m ≤ n, M devuelve la cadena vacía λ. En la entrada, m y n aparecen separados por el dígito 0.

Idea general.

Dado w = $1^m 0 1^n$, con m ≥ 0 y n ≥ 0, la MT M itera de la siguiente manera. En cada paso elimina el primer 1 del minuendo, y correspondientemente reemplaza el primer 1 del sustraendo por un 0. Al final elimina todos los 0 (caso m > n), o bien elimina todos los dígitos (caso m ≤ n).

Construcción de la MT M.

La MT M = (Q, Σ, Γ, δ, q_0, F) es tal que:

- Q = {q_0, q_1, q_2, q_3, q_4, q_5, q_6}. El estado q_0 es el estado de inicio de una iteración. El estado q_1 es el estado en que M busca el primer 0 yendo a la derecha. El estado q_2 es el estado en que M encuentra el primer 0 yendo a la derecha. El estado q_3 es el estado en que M encuentra un 1 después de un 0 yendo a la derecha. El estado q_4 es el estado en que M yendo a la derecha buscando un 1 después de un 0 encuentra en cambio un blanco. El estado q_5 es el estado en que M, iniciando una iteración, no encuentra como primer dígito un 1. El estado q_6 es el estado final.

- $\Sigma = \{1, 0\}$
- $\Gamma = \{1, 0, B\}$
- $F = \{q_6\}$
- La función de transición δ se define de la siguiente manera:

1. $\delta(q_0, 1) = (q_1, B, R)$
2. $\delta(q_1, 1) = (q_1, 1, R)$
3. $\delta(q_1, 0) = (q_2, 0, R)$
4. $\delta(q_2, 1) = (q_3, 0, L)$
5. $\delta(q_2, 0) = (q_2, 0, R)$
6. $\delta(q_3, 0) = (q_3, 0, L)$
7. $\delta(q_3, 1) = (q_3, 1, L)$
8. $\delta(q_3, B) = (q_0, B, R)$
9. $\delta(q_2, B) = (q_4, B, L)$
10. $\delta(q_4, 0) = (q_4, B, L)$
11. $\delta(q_4, 1) = (q_4, 1, L)$
12. $\delta(q_4, B) = (q_6, 1, S)$
13. $\delta(q_0, 0) = (q_5, B, R)$
14. $\delta(q_5, 0) = (q_5, B, R)$
15. $\delta(q_5, 1) = (q_5, B, R)$
16. $\delta(q_5, B) = (q_6, B, S)$

Queda como ejercicio probar que L(M) = L.

Fin de Ejemplo

Existen distintos modelos de MT equivalentes al modelo descripto previamente. Dos modelos de MT son equivalentes cuando para toda MT M_1 de un modelo existe una MT M_2 equivalente del otro, es decir que $L(M_1) = L(M_2)$. Es útil valerse de distintos modelos de MT, como se irá apreciando en el desarrollo de los temas. Presentamos a continuación algunos ejemplos, y probamos su equivalencia con el modelo inicial. Los ejemplos sirven para la ejercitación en la construcción de MT.

El primer modelo de MT que vamos a presentar se adoptará posteriormente como estándar.

Ejemplo 1.3. Máquina de Turing con estados finales q_A y q_R

Las MT M de este modelo tienen dos estados especiales, el estado q_A de aceptación y el estado q_R de rechazo. Por definición se los considera fuera del conjunto Q. Cuando M se detiene lo hace sólo en q_A o en q_R.

Formalmente, M se define como la tupla $(Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$, tal que la función de transición es

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

M acepta una entrada w cuando a partir de w 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 ejemplo, considerando el mismo lenguaje del Ejemplo 1.1, es decir $L = \{a^n b^n \mid n \geq 1\}$, se cumple que la siguiente MT $M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$ reconoce L.

El conjunto de estados Q ahora es $\{q_a, q_b, q_L, q_H\}$, no incluye el estado final q_F.

Por su parte, la función de transición δ, presentada en forma tabular, es

	a	b	α	B	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

Que el elemento de la tabla correspondiente al estado q_a y al símbolo *a*, por ejemplo, contenga la terna q_b, α, R, significa que $\delta(q_a, a) = (q_b, \alpha, R)$. Los elementos vacíos de la tabla deben entenderse como ternas con el estado de rechazo q_R.

Se cumple que el modelo inicial de MT, con un conjunto de estados finales F, y el modelo de MT con estados q_A y q_R, son equivalentes. Primero veamos que para toda MT M_1 del modelo inicial existe una MT M_2 equivalente del nuevo modelo.

Idea general.

La función δ_2 de M_2 es igual a la función δ_1 de M_1, salvo que si $\delta_1(q, x)$ no está definida se agregan las transiciones $\delta_2(q, x) = (q_A, x, S)$ o $\delta_2 (q, x) = (q_R, x, S)$, según q pertenezca o no al conjunto de estados finales F de M_1, respectivamente.

Construcción de la MT M_2.

Sea $M_1 = (Q, \Sigma, \Gamma, \delta_1, q_0, F)$. Se hace $M_2 = (Q, \Sigma, \Gamma, \delta_2, q_0, q_A, q_R)$, tal que $\forall q \in Q$, y $\forall x \in \Gamma$:

1. Si $\delta_1(q, x)$ no está definida y $q \in F$, entonces $\delta_2(q, x) = (q_A, x, S)$
2. Si $\delta_1(q, x)$ no está definida y $q \notin F$, entonces $\delta_2(q, x) = (q_R, x, S)$
3. Si $\delta_1(q, x)$ está definida, entonces $\delta_2(q, x) = \delta_1(q, x)$

Prueba de $L(M_1) = L(M_2)$.

a. Sea $w \in L(M_1)$. Entonces M_1 a partir de w se detiene en un estado $q \in F$ y un símbolo x tal que δ_1 no está definida en (q, x). Entonces por las definiciones anteriores 1 y 3, M_2 a partir de w se detiene en el estado q_A, es decir que $w \in L(M_2)$.
b. Sea $w \notin L(M_1)$. Entonces M_1 a partir de w se detiene en un estado $q \notin F$ y un símbolo x tal que δ_1 no está definida en (q, x), o no se detiene. Entonces por las definiciones 2 y 3, M_2 a partir de w se detiene en el estado q_R, o no se detiene, respectivamente, es decir que $w \notin L(M_2)$.

Veamos ahora que para toda MT M_1 del modelo con estados q_A y q_R existe una MT M_2 equivalente del modelo inicial.

Idea general.

La función de transición de M_2 es la misma que la de M_1. Lo único que cambia es que al conjunto de estados de M_2 se le agregan q_A y q_R, siendo q_A el único estado final de F.

Construcción de la MT M_2.

Sea $M_1 = (Q_1, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$. Se hace $M_2 = (Q_2, \Sigma, \Gamma, \delta, q_0, F)$, tal que $Q_2 = Q_1 \cup \{q_A, q_R\}$ y $F = \{q_A\}$.

Prueba de $L(M_1) = L(M_2)$.

a. Sea $w \in L(M_1)$. Entonces M_1 a partir de w se detiene en su estado q_A. Entonces M_2 a partir de w se detiene en su estado q_A, perteneciente a F. Entonces $w \in L(M_2)$.
b. Sea $w \notin L(M_1)$. Entonces M_1 a partir de w se detiene en su estado q_R o no se detiene. Entonces M_2 a partir de w se detiene en su estado q_R, que no pertenece a F, o no se detiene, respectivamente. Entonces $w \notin L(M_2)$.

Fin de Ejemplo

El siguiente ejemplo se utilizará en la Clase 5, cuando se presenten las máquinas RAM, las cuales constituyen un modelo de ejecución mucho más cercano a las computadoras reales.

Ejemplo 1.4. Máquina de Turing con cinta semi-infinita

La cinta de una MT de este modelo es finita a izquierda e infinita a derecha. Veamos que para toda MT M_1 con cinta infinita existe una MT M_2 equivalente con cinta semi-infinita (el sentido inverso se prueba trivialmente).

Idea general.

Supongamos que M_1 y M_2 son MT del modelo inicial, con un conjunto de estados finales. Sean $M_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, q_1, F_1)$ y $M_2 = (Q_2, \Sigma_2, \Gamma_2, \delta_2, q_2, F_2)$. Primeramente se debe introducir la noción de *track* (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 $(x_1, \ldots, x_T)$, y las 5-tuplas de la función de transición δ tienen la forma $(q_i, (x_1, \ldots, x_T), q'_i, (x'_1, \ldots, x'_T), d)$, con $d \in \{L, R, S\}$. La MT M_2 tiene 2 *tracks*. En el *track* superior simula los movimientos de M_1 a la derecha de la celda inicial, incluyéndola, y en el *track* inferior los movimientos de M_1 a la izquierda de la misma. La entrada de M_2 está al comienzo del *track* superior. Al empezar, M_2 marca su primera sub-celda inferior con el símbolo especial #. Los símbolos de Γ_2 tienen la forma (x, y). Todo símbolo x está en Γ_1, y todo símbolo *y* está en Γ_1 o es #. El estado corriente de M_2 indica si se está considerando el *track* superior o inferior. Mientras M_1 está a la derecha de su posición inicial, incluyéndola, M_2 trabaja sobre el *track superior*, moviéndose en el sentido de M_1. Y mientras M_1 está a la izquierda de su posición inicial, M_2 trabaja sobre el *track* inferior, moviéndose en el sentido opuesto al de M_1. Los estados de Q_2 son q_2 (estado inicial) más los distintos estados q^U y q^D tales que los estados q están en Q_1 (con U por *up* o arriba y D por *down* o abajo se indica si se está procesando el *track* superior o inferior, respectivamente). En particular, los estados de F_2 son los estados q^U y q^D tales que los estados q están en F_1.

Construcción de la MT M_2.

Ya se han descripto los estados y símbolos de M_2. La función de transición δ_2 se define de la siguiente manera. Para el inicio de M_2, en que hay que escribir la marca inicial #, y establecer si se procesa el *track* superior o inferior, se define:

1. Si $\delta_1(q_1, x) = (q, y, R)$, entonces $\delta_2(q_2, (x, B)) = (q^U, (y, \#), R)$.
   Si $\delta_1(q_1, x) = (q, y, S)$, entonces $\delta_2(q_2, (x, B)) = (q^U, (y, \#), S)$.

Si $\delta_1(q_1, x) = (q, y, L)$, entonces $\delta_2(q_2, (x, B)) = (q^D, (y, \#), R)$.

Para el procesamiento de M_2 cuando el símbolo corriente no tiene la marca #, se define:

2. Para todo (x, y) de Γ_2, siendo el símbolo *y* distinto de #:
   Si $\delta_1(q, x) = (p, z, R)$, entonces $\delta_2(q^U, (x, y)) = (p^U, (z, y), R)$.
   Si $\delta_1(q, x) = (p, z, L)$, entonces $\delta_2(q^U, (x, y)) = (p^U, (z, y), L)$.
   Si $\delta_1(q, x) = (p, z, S)$, entonces $\delta_2(q^U, (x, y)) = (p^U, (z, y), S)$.
   Si $\delta_1(q, y) = (p, z, R)$, entonces $\delta_2(q^D, (x, y)) = (p^D, (x, z), L)$.
   Si $\delta_1(q, y) = (p, z, L)$, entonces $\delta_{2(}q^D, (x, y)) = (p^D, (x, z), R)$.
   Si $\delta_1(q, y) = (p, z, S)$, entonces $\delta_2(q^D, (x, y)) = (p^D, (x, z), S)$.

Finalmente, para el procesamiento de M_2 cuando el símbolo corriente tiene la marca #, se define:

3. Si $\delta_1(q, x) = (p, y, R)$, entonces $\delta_2(q^U, (x, \#)) = \delta_2(q^D, (x, \#)) = (p^U, (y, \#), R)$.
   Si $\delta_1(q, x) = (p, y, S)$, entonces $\delta_2(q^U, (x, \#)) = \delta_{2(}q^D, (x, \#)) = (p^U, (y, \#), S)$.
   Si $\delta_1(q, x) = (p, y, L)$, entonces $\delta_2(q^U, (x, \#)) = \delta_2(q^D, (x, \#)) = (p^D, (y, \#), R)$.

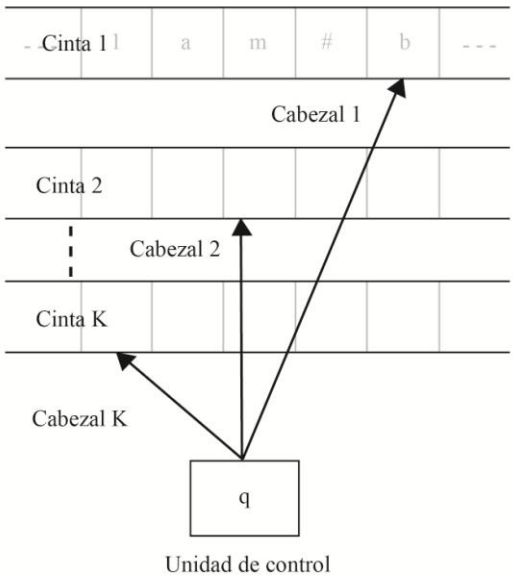
Queda como ejercicio probar que $L(M_1) = L(M_2)$.

Fin de Ejemplo

En el siguiente ejemplo se demuestra que las MT no ganan poder computacional cuando utilizan más de una cinta. Utilizaremos generalmente este modelo, con estados finales q_A y q_R, porque facilita la presentación de los ejemplos y las demostraciones de los teoremas.

Ejemplo 1.5. Máquina de Turing con varias cintas

Las MT de este modelo tienen una cantidad finita K de cintas. La cinta de entrada es siempre la primera. Cuando es necesario se especifica también una cinta de salida. Por cada cinta existe un cabezal, y sigue habiendo una sola unidad de control (ver la figura siguiente):



En la configuración inicial, la cinta 1 contiene la entrada, la unidad de control almacena el estado q_0, el cabezal de la cinta 1 (cabezal 1) apunta al primer símbolo de la entrada, y el resto de los cabezales apuntan a alguna celda de las cintas correspondientes, al comienzo todas en blanco. Luego la MT se comporta de acuerdo a su función de transición δ. En cada paso lee un estado y una K-tupla de símbolos, los apuntados, respectivamente, por los cabezales 1 a K, eventualmente los modifica, y 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, asumiendo el uso de los estados q_A y q_R, se define $M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$, tal que

$$\delta: Q \times \Gamma^K \rightarrow (Q \cup \{q_A, q_R\}) \times (\Gamma \times \{L, R, S\})^K$$

Por ejemplo, sea $L = \{w \mid w \in \{a, b\}^* \text{ y } w \text{ es un palíndrome}\}$. Una cadena w es un palíndrome si y sólo si $w = w^R$, siendo w^R la cadena inversa de w. Se va a construir una MT M con 2 cintas que acepta L.

Idea general.

La MT M copia la entrada desde la cinta 1 a la cinta 2. Apunta al primer símbolo de la cadena de la cinta 1 y al último símbolo de la cadena de la cinta 2. E itera de la siguiente

manera: compara los símbolos apuntados; si son distintos rechaza; si son blancos acepta; y en otro caso se mueve a la derecha en la cinta 1, a la izquierda en la cinta 2, y vuelve a la comparación de símbolos.

Construcción de la MT M.

Se define $M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$, con:

- $Q = \{q_0, q_1, q_2\}$. El estado q_0 es el estado de copia de la entrada desde la cinta 1 a la cinta 2. El estado q_1 es el estado de posicionamiento de los cabezales 1 y 2 luego de la copia. Y el estado q_2 es el estado de comparación de las cadenas de las cintas 1 y 2.
- $\Sigma = \{a, b\}$
- $\Gamma = \{a, b, B\}$
- $q_0 = q_0$
- La función de transición δ es

	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

Que el elemento de la tabla correspondiente al estado q_0 y al par de símbolos a, B, por ejemplo, contenga la 5-tupla q_0, a, R, a, R, significa que $\delta(q_0, (a, B)) = (q_0, (a, R), (a, R))$, simplificando paréntesis. Queda como ejercicio probar que L(M) = L.

Veamos ahora que para toda MT M_1 con K cintas, existe una MT M_2 equivalente con una sola cinta (el sentido inverso se prueba trivialmente).

Idea general.

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, 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 blancos en todas las sub-celdas salvo en una que lleva la marca C, para indicar que la celda representada está apuntada por un cabezal. La tabla siguiente ejemplifica la representación descripta, para el caso de 3 cintas y por lo tanto 6 *tracks*:

	i	i + 1	i + 2	i + 3	i + 4	i + 5
1	…	x_1	x_4	x_7	x_{10}	…
2	…	C				…
3	…	x_2	x_5	x_8	x_{11}	…
4	…				C	…
5	…	x_3	x_6	x_9	x_{12}	…
6	…			C		…

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 tiene el símbolo x_5 y no está apuntada por el cabezal 2, etc. Si la entrada de M_1 es $w = w_1 \ldots w_n$, entonces al comienzo la entrada de M_2 es $(w_1, B, \ldots, B) \ldots (w_n, B, \ldots, B)$, y el cabezal apunta al símbolo $(w_1, B, \ldots, B)$. M_2 empieza transformando el primer símbolo de su entrada en el símbolo $(w_1, C, B, C, \ldots, B, C)$, y luego pasa a un estado que simula el estado inicial de M_1. A partir de acá, 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, de la siguiente manera:

1. Al inicio de este conjunto de pasos de M_2 , su cabezal apunta a la celda con la marca C de más a la izquierda.
2. Luego M_2 se mueve a la derecha, memorizando por medio de sus estados uno a uno 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 incluirá 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 el símbolo asociado que representa el contenido de M_1, y 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.

Queda como ejercicio construir M_2 y probar que $L(M_1) = L(M_2)$. Notar 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 pasos de M_1 a lo sumo unos $(4 + 8 + 12 + \ldots + 4h) = 4(1 + 2 + 3 + \ldots + h) = O(h^2)$ pasos. Esto significa que si bien la cantidad de cintas no influye en el poder computacional de una MT, sí impacta en su tiempo de trabajo. El tiempo de retardo cuando se reduce de K cintas a una cinta, independientemente del valor de K, es del orden cuadrático en el peor caso. Esta relación la utilizaremos cuando tratemos la complejidad computacional temporal, en la segunda parte del libro.

Fin de Ejemplo

En el ejemplo anterior se muestra cómo un estado puede almacenar información. A continuación se ejemplifica cómo una MT puede simular los estados de otra máquina por medio de los símbolos de su alfabeto.

Ejemplo 1.6. Máquina de Turing con un solo estado

Vamos a probar que para toda MT M_1 con varios estados existe una MT M_2 equivalente con un solo estado (el sentido inverso se prueba trivialmente).

Idea general.

Supongamos que M_1 tiene una cinta y además de q_A y q_R, los estados $q_0, \ldots, q_m$. Con una MT M_2 con dos cintas y un solo estado q, además de q_A y q_R, se puede simular M_1. M_2 simula M_1 en la cinta 1, y en la cinta 2 representa con símbolos nuevos $x_0, \ldots, x_m$,

los estados de M_1. Al comienzo, M_2 escribe en la cinta 2 el símbolo x_0 que representa el estado inicial q_0 de M_1, y luego simula paso a paso M_1, haciendo lo mismo salvo que en lugar de cambiar de estado cambia de símbolo en la cinta 2.

Construcción de la MT M_2.

Sea $M_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_0, q_A, q_R)$ con una cinta y estados $q_0, \ldots, q_m$. Se hace $M_2 = (\{q\}, \Sigma, \Gamma_1 \cup \{x_0, \ldots, x_m\}, \delta_2, q, q_A, q_R)$ con dos cintas. La función de transición δ_2 se define de la siguiente manera:

1. $\forall x \in \Sigma$: $\delta_2(q, (x, B)) = (q, (x, S), (x_0, S))$.
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 distinto de q_A y 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. Lo mismo que 3 pero considerando q_R en lugar de q_A.

Queda como ejercicio probar que $L(M_1) = L(M_2)$.

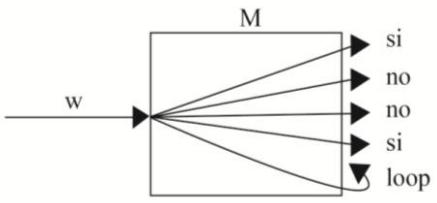
Fin de Ejemplo

El siguiente es un último ejemplo de modelo equivalente de MT, que se utilizará fundamentalmente en la segunda parte del libro. En él se demuestra que las MT tampoco ganan poder computacional cuando son *no determinísticas*, es decir cuando a partir de un par (q, x) pueden tener varias transiciones. En la clase que viene se muestra un ejemplo de uso de este modelo.

Ejemplo 1.7. Máquina de Turing no determinística

La elección de por cuál terna (q', x', d) a partir de (q, x) la MT no determinística (o MTN) continúa, es como el nombre del modelo lo indica no determinística. Ahora se considera una *relación de transición* Δ, en lugar de una función de transición δ. Una manera de interpretar cómo trabaja una MTN M es suponer que todas sus computaciones o secuencias de pasos se ejecutan en paralelo. Asumiendo que tiene los estados q_A y q_R, M acepta una entrada w si a partir de w al menos una computación se detiene en q_A. En caso contrario, es decir si todas las computaciones de M terminan en

el estado q_R o son infinitas, M rechaza w. La siguiente figura ilustra un posible comportamiento de una MTN M:



En la figura, los términos sí, no y *loop*, indican que la computación correspondiente se detiene en q_A, se detiene en q_R, o no se detiene, respectivamente. En este caso la MTN acepta la entrada w porque al menos una de sus computaciones la acepta. Formalmente, asumiendo una sola cinta, se define $M = (Q, \Sigma, \Gamma, \Delta, q_0, q_A, q_R)$, con

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

siendo $P((Q \cup \{q_A, q_R\}) \times \Gamma \times \{L, R, S\})$ 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 $\Delta(q, x) = \varnothing$ establece que Δ no está definida para (q, x).

Una MT determinística (o MTD) es un caso particular de MTN, tal que su relación de transición tiene grado 1. Se demuestra a continuación que para toda MTN M_1 existe una MTD M_2 equivalente.

Idea general.

Para todo estado y todo símbolo de una MTN M_1, existe un número finito de posibles siguientes pasos, digamos las alternativas 1, …, K, siendo K el grado de la relación de transición Δ. De esta manera, se puede representar cualquier computación de M_1 mediante una secuencia de dígitos entre 1 y K, que denominaremos *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 alternativa de Δ, en el segundo paso la cuarta alternativa, en el tercero la primera, etc. Algunos discriminantes pueden representar computaciones no válidas, dado que no siempre hay K alternativas para un par (q, x).

Se puede construir una MTD M_2 equivalente a M_1 con tres cintas. La primera cinta tiene la entrada. En la segunda cinta M_2 genera sistemáticamente los discriminantes, de menor a mayor longitud y en orden numérico creciente considerando la misma longitud. Por ejemplo, los primeros discriminantes son: (1), …, (K), (1, 1), …, (1, K), …, (K, 1), …, (K, K), (1, 1, 1), …, (1, 1, K), … Este orden se denomina *orden lexical canónico*, o simplemente *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 la entrada 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 alternativa no válida en el discriminante, M_2 genera el siguiente según el orden canónico. Si M_1 acepta, entonces M_2 acepta (M_2 solamente acepta o no se detiene).

La simulación efectuada por la MTD M_2 consiste en recorrer a lo ancho, empleando la técnica de *breadth first search*, el árbol de computaciones asociado a la MTN M_1, en el sentido de que primero simula un paso de todas las computaciones de M_1, después dos, después tres, etc. Naturalmente no sirve recorrer el árbol de computaciones a lo largo, es decir rama por rama, empleando la técnica de *depth first search*, porque al simular una rama infinita de M_1 antes que una de aceptación, M_2 nunca podrá ejecutar esta última.

Construcción de la MTD M_2.

La parte más compleja de la construcción consiste en transformar la relación de transición Δ en una función de transición δ. Básicamente, si por ejemplo $\Delta(q, x) = \{(q_1, x_1, d_1), (q_2, x_2, d_2)\}$, entonces δ tendría transiciones del tipo $\delta(q, (y, 1, x)) = (q_1, (y, S), (1, R), (x_1, d_1))$ y $\delta(q, (y, 2, x)) = (q_2, (y, S), (2, R), (x_2, d_2))$. La construcción de M_2 queda como ejercicio.

Prueba de $L(M_1) = L(M_2)$.

a. Si $w \in L(M_1)$, entonces alguna computación de la MTN M_1 acepta w. Por construcción, M_2 acepta w la vez que el discriminante asociado a dicha computación se genera en su cinta 2. Por lo tanto, $w \in L(M_2)$.
b. Si $w \in L(M_2)$, entonces M_2 acepta w a partir de 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, $w \in L(M_1)$.

Notar que si K es el grado de la relación de transición Δ de M_1, h pasos de M_1 se simulan 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 no determinismo no le agrega poder computacional a las MT, el tiempo de ejecución sí se impacta cuando se simula determinísticamente una MTN. El tiempo de retardo es del orden exponencial en el peor caso. Al igual que lo que comentamos cuando describimos las MT con varias cintas, utilizaremos esta relación cuando estudiemos la complejidad computacional temporal en la segunda parte del libro.

Fin de Ejemplo

Ejercicios de la Clase 1

1. Completar la prueba del Ejemplo 1.1.
2. Completar la prueba del Ejemplo 1.2.
3. Construir una MT distinta a la del Ejemplo 1.2 para restar dos números naturales, que consista en eliminar primero el primer uno del minuendo y el último uno del sustraendo, luego el segundo uno del minuendo y el anteúltimo uno del sustraendo, y así sucesivamente.
4. Construir una MT que reconozca $L = \{a^n b^n c^n \mid n \geq 0\}$ de la manera más eficiente posible con respecto al número de pasos.
5. Construir una MT que reconozca el lenguaje de las cadenas de unos y ceros con igual cantidad de ambos símbolos.
6. Construir una MT M que reconozca:
   i. $L = \{x\#y\#z \mid z = x + y\}$.
   ii. $L = \{x\#y \mid y = 2^x\}$.
   iii. $L = \{x\#y \mid y = x!\}$.

   En todos los casos, x, y, z, son números naturales representados en notación binaria.
7. Construir una MT que genere todas las cadenas de $\{0,1\}^*$ en orden canónico.
8. Completar la prueba del Ejemplo 1.4.
9. Completar las pruebas del Ejemplo 1.5.
10. Completar la prueba del Ejemplo 1.6.
11. Completar la prueba del Ejemplo 1.7.
12. Modificar la MTD planteada en el Ejemplo 1.7 para simular una MTN, de manera que ahora se detenga si detecta que todas las computaciones de la MTN se detienen.

13. Probar que dada una MTN, existe otra MTN equivalente cuya relación de transición tiene grado dos.
14. Sea USAT el lenguaje de las fórmulas booleanas con exactamente una asignación de valores de verdad que las satisface. La sintaxis precisa de las fórmulas booleanas se define en la segunda parte del libro; por lo pronto, asumir que poseen variables y operadores lógicos como $\neg$, $\vee$ y $\wedge$. Por ejemplo, $(x_1 \wedge \neg x_4) \vee x_2$ es una fórmula booleana, y la asignación con las tres variables verdaderas satisface la fórmula. Se pide determinar si la siguiente MTN reconoce USAT:
    1. Si la entrada no es una fórmula booleana correcta sintácticamente, rechaza.
    2. Genera no determinísticamente una asignación de valores de verdad A, y si A no satisface la fórmula, rechaza.
    3. Genera no determinísticamente una asignación de valores de verdad A' $\neq$ A. Si A' no satisface la fórmula, acepta, y si A' la satisface, rechaza.
15. Probar que todo modelo de MT presentado a continuación es equivalente a alguno de los que hemos descripto en la Clase 1:
    i. No tienen el movimiento S.
    ii. Al comienzo no se sabe a qué celdas apuntan los cabezales.
    iii. Cuando aceptan, todas las cintas tienen únicamente símbolos blancos.
    iv. Tienen sólo el movimiento R y el movimiento JUMP, tal que el efecto del JUMP es posicionarse sobre el símbolo de más a la izquierda.
    v. Escriben en una celda a lo sumo una vez.
    vi. Pueden remover e insertar celdas.
    vii. Tienen varios cabezales por cinta.
    viii. El cabezal se mueve de extremo a extremo del contenido de la única cinta, de izquierda a derecha, de derecha a izquierda, y así sucesivamente (se permite el uso del movimiento S).
16. Las *funciones recursivas primitivas* constituyen un importante paso en la formalización de la noción de computabilidad. Se definen utilizando como principales operaciones la recursión y la composición, y forman un subconjunto propio de las funciones recursivas (parciales), que son precisamente las funciones computables. Las funciones recursivas se definen agregando el operador de búsqueda no acotada, que permite definir funciones parciales. Un ejemplo muy conocido de función recursiva que no es recursiva primitiva es la función de Ackermann. Muchas de las funciones normalmente estudiadas en la teoría de

números son recursivas primitivas. Como ejemplos están la suma, la división, el factorial, etc. El argumento de una función recursiva primitiva es un número natural o una n-tupla de números naturales (i_1, i_2, ..., i_n), y el valor es un número natural. Las funciones recursivas primitivas se definen según las siguientes reglas:

1. Para todo $k \geq 0$, la función cero k-aria definida como $cero_k(n_1, ..., n_k) = 0$, para todo número natural n_1, ..., n_k, es recursiva primitiva.
2. La función sucesor S, de aridad 1, que produce el siguiente número natural, es recursiva primitiva.
3. Las funciones de proyección P_i^n, de aridad n, que producen como valor el argumento de la posición i, son recursivas primitivas.
4. Dada una función recursiva primitiva f de aridad k, y funciones recursivas primitivas g_1, ..., g_k, de aridad n, la composición de f con g_1, ..., g_k, es decir la función $h(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_k(x_1, ..., x_n))$, es recursiva primitiva.
5. Dada una función recursiva primitiva f de aridad k, y una función recursiva primitiva g de aridad k + 2, la función h de aridad k + 1 definida como $h(0, x_1, ..., x_k) = f(x_1, ..., x_k)$ y $h(S(n), x_1, ..., x_k) = g(h(n, x_1, ..., x_k), n, x_1, ..., x_k)$, es recursiva primitiva.

Así, una función es recursiva primitiva si es la función constante cero, la función sucesor, una proyección, o si se define a partir de funciones recursivas primitivas utilizando únicamente composición y recursión. Por ejemplo la suma, que se comporta de la siguiente manera: suma(0, x) = x, y suma(n + 1, x) = suma(n, x) + 1, llevada al esquema de las funciones recursivas primitivas se puede definir así: $suma(0, x) = P_1^1(x)$, y $suma(S(n), x) = S(P_1^3(suma(n, x), n, x))$.

Se pide probar que toda función recursiva primitiva f es una función total computable, es decir que existe una MT que a partir de cualquier cadena w computa f(w) y se detiene.

Clase 2. Jerarquía de la computabilidad

Hay una jerarquía de clases de lenguajes (o lo que es lo mismo, de problemas de decisión), teniendo en cuenta si son reconocidos y de qué manera por una máquina de Turing. En esta clase describimos dicha jerarquía de la computabilidad, y probamos algunas propiedades de las clases de lenguajes que la componen. Para facilitar las demostraciones desarrolladas, las máquinas de Turing utilizadas se describen con mayor nivel de abstracción que en la clase anterior.

Definición 2.1. Lenguajes recursivamente numerables y recursivos

Un lenguaje es *recursivamente numerable* si y sólo si existe una MT que lo reconoce.
Es decir, si $\mathfrak{L}$ es el conjunto de todos los lenguajes (cada uno integrado por cadenas finitas de símbolos pertenecientes a un alfabeto universal Σ), sólo los lenguajes recursivamente numerables de $\mathfrak{L}$ son reconocibles por una MT (por esto es que a los problemas de decisión asociados se los conoce como *computables*). La clase de los lenguajes recursivamente numerables se denomina RE (por *recursively enumerable languages*). El nombre se debe a que las cadenas de estos lenguajes se pueden enumerar. De esta manera, dado $L \in RE$, si M es una MT tal que $L(M) = L$, se cumple para toda cadena w de Σ^* que:

- Si $w \in L$, entonces M a partir de w se detiene en su estado q_A.
- Si $w \notin L$, entonces M a partir de w se detiene en su estado q_R o no se detiene.

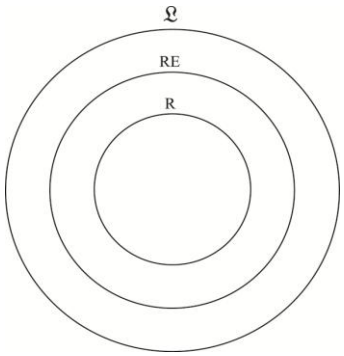
Se va a probar después que no todos los lenguajes son recursivamente numerables, y que sólo algunos tienen la propiedad de que las MT que los reconocen se detienen siempre. Considerando este último caso, se define que un lenguaje es *recursivo* si y sólo si existe una MT M que lo reconoce y que se detiene cualquiera sea su entrada. La clase de los lenguajes recursivos se denomina R. A los problemas de decisión asociados se los conoce como *decidibles*, porque las MT que los resuelven pueden justamente decidir, cualquiera sea la instancia, si es positiva o negativa. Ahora, dado $L \in R$, si M es una MT tal que $L(M) = L$, se cumple para toda cadena w de Σ^* que:

- Si $w \in L$, entonces M a partir de w se detiene en su estado q_A.

- Si $w \notin L$, entonces M a partir de w se detiene en su estado q_R.

Fin de Definición

Se cumple por definición que $R \subseteq RE \subseteq \mathfrak{L}$. Probaremos entre esta clase y la que viene que $R \subset RE \subset \mathfrak{L}$, es decir que no todos los problemas computables son decidibles, y que no todos los problemas son computables (las fronteras de R y RE determinan los límites de la decidibilidad y la computabilidad, respectivamente). Esto se ilustra en la siguiente figura, que presenta una primera versión de la jerarquía de la computabilidad:



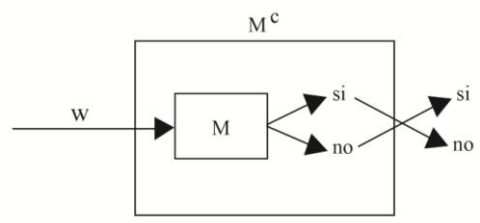
En los dos teoremas que desarrollamos a continuación se presentan algunas propiedades de clausura de las clases R y RE. Algunos de estos resultados se utilizan en la prueba de correctitud de la jerarquía planteada, al tiempo que sus demostraciones permiten continuar con la ejercitación en la construcción de MT, ahora utilizando notación algorítmica en lugar de funciones de transición, e incorporando ejemplos de composición de MT, técnica que se seguirá empleando en las próximas clases.

Teorema 2.1. Algunas propiedades de clausura de la clase R

Considerando las operaciones de complemento, intersección, unión y concatenación de lenguajes, se cumple que la clase R es cerrada con respecto a todas ellas. Vamos a probar primeramente que la clase R es cerrada con respecto al complemento. Dado un lenguaje L, su lenguaje complemento es $L^C = \{w \mid w \in \Sigma^* \wedge w \notin L\}$. Demostramos a continuación que si $L \in R$, entonces también $L^C \in R$.

Idea general.

Dado un lenguaje recursivo L, sea M una MT que lo acepta y se detiene siempre, es decir a partir de cualquier entrada. Se va a construir una MT M^C que acepta L^C y se detiene siempre, de la siguiente manera: dada una entrada w, si M se detiene en q_A, entonces M^C se detiene en q_R, y viceversa. La figura siguiente ilustra esta idea:



Construcción de la MT M^C.

Si $M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$, entonces $M^C = (Q, \Sigma, \Gamma, \delta', q_0, q_A, q_R)$, con δ y δ' idénticas salvo en la aceptación y rechazo. Para todo estado q de Q, todo par de símbolos x_i y x_k de Γ, y todo movimiento d del conjunto {L, R, S}, se define:

1. Si $\delta(q, x_i) = (q_A, x_k, d)$, entonces $\delta'(q, x_i) = (q_R, x_k, d)$
2. Si $\delta(q, x_i) = (q_R, x_k, d)$, entonces $\delta'(q, x_i) = (q_A, x_k, d)$

Prueba de que M^C se detiene siempre.

a. $w \in L^C \rightarrow w \notin L \rightarrow$ con entrada w, M se detiene en $q_R \rightarrow$ con entrada w, M^C se detiene en q_A.
b. $w \notin L^C \rightarrow w \in L \rightarrow$ con entrada w, M se detiene en $q_A \rightarrow$ con entrada w, M^C se detiene en q_R.

Prueba de $L(M^C) = L^C$.

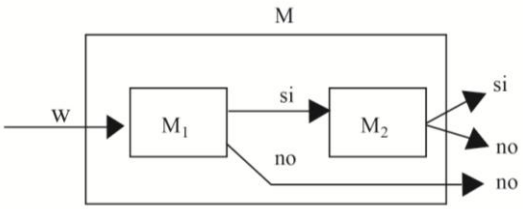
$w \in L(M^C) \leftrightarrow$ con entrada w, M^C se detiene en $q_A \leftrightarrow$ con entrada w, M se detiene en $q_R \leftrightarrow w \notin L \leftrightarrow w \in L^C$.

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 L. La prueba por construcción de que un lenguaje L es recursivamente numerable, en cambio, requiere solamente la construcción de una MT M y la prueba de que M acepta L.

La clase R también es cerrada con respecto a las operaciones de intersección y unión de lenguajes. Es decir, si $L_1 \in R$ y $L_2 \in R$, entonces $L_1 \cap L_2 \in R$, y $L_1 \cup L_2 \in R$. Vamos a probar el caso de la intersección. El caso de la unión queda como ejercicio.

Idea general.

Sean M_1 una MT que acepta L_1 y se detiene siempre, y M_2 una 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, con las siguientes características. M simula primero M_1 y luego M_2. Dada una entrada 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 la misma entrada 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 siguiente ilustra esta idea:



Construcción de la MT M.

M tiene dos cintas. Dada una entrada w en la cinta 1, M hace:

1. Copia w en la cinta 2.
2. Simula 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. Borra el contenido de la cinta 2.
4. Copia w en la cinta 2.
5. Simula 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).

Queda como ejercicio probar que M se detiene siempre, y que $L(M) = L_1 \cap L_2$. Notar que a diferencia de las simulaciones presentadas en la clase anterior, ahora de lo que se trata es de ejecutar directamente una MT M' por parte de otra MT M. M "invoca a la

subrutina" M', o en otras palabras, la función de transición δ de M incluye un fragmento δ' que no es sino la función de transición de M'. En la clase siguiente, en que se trata la máquina de Turing universal, este concepto se formaliza.

Otra propiedad de la clase R que demostramos a continuación es que es cerrada con respecto a la concatenación de lenguajes, es decir que si $L_1 \in R$ y $L_2 \in R$, entonces también $L_1 \cdot L_2 \in R$, siendo $L_1 \cdot L_2 = \{w \mid \exists w_1, \exists w_2, w_1 \in L_1, w_2 \in L_2, w = w_1 w_2\}$.

Idea general.

Sean M_1 una MT que acepta L_1 y se detiene siempre, y M_2 una MT que acepta L_2 y se detiene siempre. Se va a construir una MT M que acepta $L_1 \cdot L_2$ y se detiene siempre, con las siguientes características. Dada una entrada w, con $|w| = n$, M simula M_1 a partir de los primeros 0 símbolos de w, y M_2 a partir de los últimos n símbolos de w, y si en ambos casos hay aceptación, entonces M acepta w. En caso contrario, M hace lo mismo pero ahora con el primer símbolo de w y los últimos n – 1 símbolos de w. Mientras no se detenga por aceptación, M repite el proceso con 2 y n – 2 símbolos de w, 3 y n – 3 símbolos, y así siguiendo hasta llegar a los n y 0 símbolos, en cuyo caso M rechaza w.

Construcción de la MT M.

M tiene cinco cintas. A partir de una entrada w en su cinta 1, tal que $|w| = n$, hace:

1. Escribe el número 0 en la cinta 2. Sea i dicho número.
2. Escribe el número n en la cinta 3. Sea k dicho número.
3. Escribe los primeros i símbolos de w en la cinta 4.
4. Escribe los últimos k símbolos de w en la cinta 5.
5. Simula M_1 en la cinta 4 a partir del contenido de dicha cinta, y simula M_2 en la cinta 5 a partir del contenido de dicha cinta. Si ambas simulaciones se detienen en q_A, entonces M se detiene en q_A.
6. Si $i = n$, se detiene en q_R.
7. Hace $i := i + 1$ en la cinta 2, $k := k - 1$ en la cinta 3, borra los contenidos de las cintas 4 y 5, y vuelve al paso 3.

Queda como ejercicio probar que M se detiene siempre y que $L(M) = L_1 \cdot L_2$.

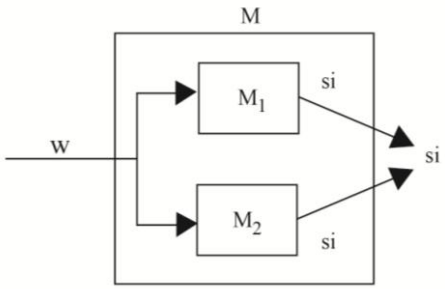
Fin de Teorema

Teorema 2.2. Algunas propiedades de clausura de la clase RE

Considerando las operaciones de intersección, unión y concatenación de lenguajes, se cumple que también la clase RE es cerrada con respecto a ellas. En cambio, a diferencia de la clase R, RE no es cerrada con respecto al complemento, lo que se probará más adelante. Vamos a demostrar primero que RE es cerrada con respecto a la unión de lenguajes, es decir que si $L_1 \in RE$ y $L_2 \in RE$, entonces $L_1 \cup L_2 \in RE$. La prueba de que RE es cerrada con respecto a la intersección queda como ejercicio.

Idea general.

Sean M_1 una MT que acepta L_1 y M_2 una 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 $L_1 \cup L_2$. No sirve que la MT M simule primero M_1 y luego M_2, porque de esta manera M no acepta las cadenas de M_2 a partir de las que M_1 no se detiene. El mismo problema ocurre simulando primero M_2 y después M_1. La solución es otra: se simulan "en paralelo" las MT M_1 y M_2, y se acepta si alguna de las dos MT acepta. La figura siguiente ilustra esta idea:



Construcción de la MT M.

Sea la siguiente MT M con cuatro cintas. Dada una entrada w en la cinta 1, M hace:

1. Copia w en las cintas 2 y 3.
2. Escribe el número 1 en la cinta 4. Dicho número se va a referenciar como i en lo que sigue.
3. Simula, 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 o M_2 se detienen en q_A, entonces M se detiene en q_A.
4. Borra el contenido de las cintas 2 y 3.

5. Copia w en las cintas 2 y 3.
6. Suma 1 al número i de la cinta 4.
7. Vuelve al paso 3.

Notar que la MT M se detiene en su estado q_A o no se detiene. Podría modificarse la construcción, haciendo que se detenga en q_R cuando 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 las dos MT desde el principio. En el paso 3 se indica que se simulan a lo sumo i pasos porque M_1 y M_2 pueden detenerse antes. La implementación de estas simulaciones en términos de la función de transición de M podría consistir, asumiendo una representación en base unaria del contador i, en avanzar sobre él a la derecha un dígito por cada paso simulado de M_1 y M_2. Queda como ejercicio probar que $L(M) = L_1 \cup L_2$.

Una construcción alternativa, valiéndonos de las MT no determinísticas (o MTN) descriptas en la clase anterior, es la siguiente. Si $M_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, q_{10}, q_A, q_R)$ y $M_2 = (Q_2, \Sigma_2, \Gamma_2, \delta_2, q_{20}, q_A, q_R)$ son dos MT determinísticas (o MTD) que aceptan los lenguajes L_1 y L_2, respectivamente, se puede construir una MTN M que acepta $L_1 \cup L_2$ a partir de M_1 y M_2. Sea q_0 un estado que no está en Q_1 ni en Q_2. La MTN M es la tupla $M = (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma_1 \cup \Sigma_2, \Gamma_1 \cup \Gamma_2, \Delta, q_0, q_A, q_R)$, tal que

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

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

Al igual que la clase R, RE es cerrada con respecto a la concatenación de lenguajes, es decir que si $L_1 \in RE$ y $L_2 \in RE$, entonces también $L_1 \cdot L_2 \in RE$, lo que se prueba a continuación. Como en el caso de la unión de lenguajes recursivamente numerables visto recién, debe tenerse en cuenta que las MT consideradas pueden no detenerse en caso de rechazo.

Idea general.

Tal como se hizo con los lenguajes recursivos, se va a construir una MT M que reconozca $L_1 \cdot L_2$ simulando M_1 y M_2 (las MT que reconocen L_1 y L_2, respectivamente),

primero a partir de 0 y |w| símbolos de la entrada w, después a partir de 1 y |w – 1| símbolos, y así siguiendo hasta llegar a |w| y 0 símbolos, aceptando eventualmente. La diferencia con el caso de los lenguajes recursivos está en que ahora, teniendo en cuenta las posibles no detenciones de M_1 y M_2, M debe simularlas "en paralelo". La MT M primero hace simulaciones de un paso de M_1 y M_2 con todas las posibles particiones de la entrada w (|w| + 1 posibilidades), luego hace simulaciones de a lo sumo dos pasos, y así siguiendo hasta eventualmente aceptar (éste es el caso en que al cabo de a lo sumo un determinado número de pasos, digamos k, M_1 acepta los primeros i símbolos de w y M_2 acepta los últimos |w| – i símbolos de w).

Construcción de la MT M.

Sea la siguiente MT M con seis cintas, que a partir de una entrada w en su cinta 1, tal que |w| = n, hace:

1. Escribe el número 1 en la cinta 2. Sea h dicho número.
2. Escribe el número 0 en la cinta 3. Sea i dicho número.
3. Escribe el número n en la cinta 4. Sea k dicho número.
4. Escribe los primeros i símbolos de w en la cinta 5.
5. Escribe los últimos k símbolos de w en la cinta 6.
6. Simula a lo sumo h pasos de M_1 en la cinta 5 a partir del contenido de dicha cinta, y simula a lo sumo h pasos de M_2 en la cinta 6 a partir del contenido de dicha cinta. Si ambas simulaciones se detienen en q_A, entonces M se detiene en q_A.
7. Si i = n, hace h := h + 1 en la cinta 2, borra los contenidos de las cintas 3, 4, 5 y 6, y vuelve al paso 2.
8. Hace i := i + 1 en la cinta 3, k := k – 1 en la cinta 4, borra los contenidos de las cintas 5 y 6, y vuelve al paso 4.

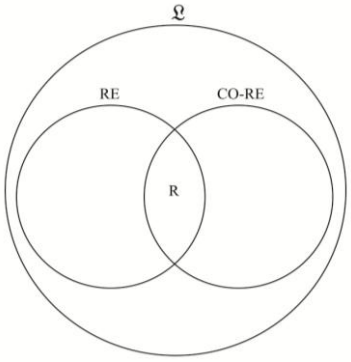
Queda como ejercicio probar que $L(M) = L_1 \cdot L_2$.

Fin de Teorema

En lo que sigue de esta clase hasta el final, empezamos a justificar formalmente la estructura de la jerarquía de la computabilidad presentada. En la clase que viene

completamos la prueba (por razones de claridad en la exposición de los temas, posponemos hasta la Clase 3 la demostración de que $R \subset RE$).

Sea CO-RE la clase de los lenguajes complemento, con respecto a Σ^*, de los lenguajes recursivamente numerables. Formalmente: $CO\text{-}RE = \{L \mid L \in \mathfrak{L} \wedge L^C \in RE\}$. Considerando CO-RE, la siguiente figura muestra una versión más detallada de la jerarquía de la computabilidad:



De la figura se desprende que un lenguaje L es recursivo si y sólo si tanto L como L^C son recursivamente numerables, lo que se prueba a continuación.

Teorema 2.3. R = RE ∩ CO-RE

Se prueba fácilmente que $R \subseteq RE \cap CO\text{-}RE$.

La inclusión $R \subseteq RE$ se cumple por definición.

También vale $R \subseteq CO\text{-}RE$ porque $L \in R \rightarrow L^C \in R \rightarrow L^C \in RE \rightarrow L \in CO\text{-}RE$.

Veamos que también se cumple la inversa, es decir, $RE \cap CO\text{-}RE \subseteq R$.

Idea general.

Sean M y M^C dos MT que aceptan los lenguajes L y L^C, respectivamente.

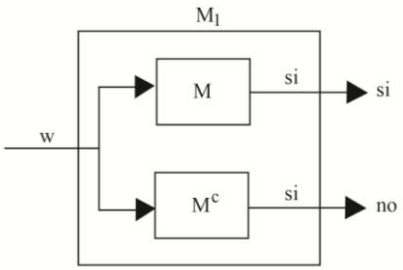
Se va a construir una MT M_1 que se detiene siempre y acepta L, de la siguiente manera.

M_1 simula "en paralelo" las MT M y M^C.

Si M se detiene en q_A, entonces M_1 se detiene en su estado q_A.

Y si M^C se detiene en q_A, entonces M_1 se detiene en su estado q_R.

La figura siguiente ilustra esta idea:



Se cumple que M_1 se detiene siempre porque para toda entrada w, vale que $w \in L$ o $w \in L^C$, y por lo tanto M o M^C aceptan w. Queda como ejercicio la construcción de M_1 y la prueba de que $L(M_1) = L$.

Fin de Teorema

En la última figura que ilustra la jerarquía de la computabilidad, se distinguen cuatro categorías de lenguajes. Enumeradas de acuerdo a su dificultad creciente, son:

1. R
2. RE – R
3. CO-RE – R
4. $\mathfrak{L}$ – (RE $\cup$ CO-RE)

En este contexto, dado un par cualquiera de lenguajes L y L^C, se cumple alternativamente que:

- Tanto L como L^C pertenecen a R.
- L pertenece a RE – R, y L^C pertenece a CO-RE – R.
- Tanto L como L^C pertenecen a $\mathfrak{L}$ – (RE $\cup$ CO-RE).

Por el Teorema 2.1, si L está en R también lo está L^C. Hemos visto en la clase anterior dos ejemplos de lenguajes recursivos, los lenguajes $\{a^n b^n \mid n \geq 1\}$ y {w | w $\in$ {a, b}* y w es un palíndrome} (ver, respectivamente, el Ejemplo 1.1 y el Ejemplo 1.5).
En la próxima clase vamos a encontrar un primer ejemplo de lenguaje L en RE – R, con lo que probaremos que R $\subset$ RE, o en otras palabras, que hay problemas computables que no son decidibles. En este caso L^C pertenece a CO-RE – R, de acuerdo también al Teorema 2.1. Notar que con la existencia de L^C se prueba además que RE $\subset$ $\mathfrak{L}$, de

acuerdo al Teorema 2.3, o en otras palabras, que hay problemas que no son computables. Adelantándonos, el primer lenguaje de RE – R que presentaremos corresponde al problema de la detención de las MT, o directamente el problema de la detención (también conocido como *halting problem*). Este problema computable no decidible aparece en la ya referida publicación de A. Turing en la que presenta las máquinas que llevan su nombre. A modo de ilustración, otros ejemplos clásicos de problemas computables no decidibles, esta vez no relacionados con MT, son (sobre algunos de ellos volveremos más adelante):

- El problema de la validez en la lógica de primer orden (o cálculo de predicados), es decir el problema de determinar si una fórmula de la lógica de primer orden es válida, o lo que es lo mismo, si es un teorema (el trabajo mencionado de Turing se relaciona con este problema).
- El problema de correspondencia de Post, definido de la siguiente manera: dada una secuencia de pares de cadenas de unos y ceros no vacías $(s_1, t_1), \ldots, (s_k, t_k)$, determinar si existe una secuencia de índices $i_1, \ldots, i_n$, con $n \geq 1$, tal que las cadenas $s_{i1} \ldots s_{in}$ y $t_{i1} \ldots t_{in}$ sean iguales.
- El problema de determinar si una ecuación diofántica (ecuación algebraica con coeficientes y soluciones enteras) tiene solución.
- El problema de teselación del plano: dado un conjunto finito de formas poligonales, determinar si con ellas se puede cubrir el plano.
- El problema de las palabras para semigrupos: dadas dos cadenas s, t, y un conjunto finito de igualdades entre cadenas del tipo $s_1 = t_1, \ldots, s_k = t_k$, determinar si de la cadena s se puede llegar a la cadena t por medio de sustituciones de subcadenas empleando las igualdades definidas.

Nos queda tratar el caso en que tanto L como L^C pertenecen a $\mathfrak{L} - (RE \cup CO\text{-}RE)$, lo que prueba que $RE \cup CO\text{-}RE \subset \mathfrak{L}$. En el ejemplo siguiente presentamos un lenguaje de estas características, un tanto artificial. Luego veremos lenguajes más naturales de este tipo. A modo de ilustración, un caso clásico de lenguaje no recursivamente numerable tal que su complemento tampoco lo es, es el de los enunciados verdaderos de la teoría de números (o aritmética), de acuerdo al Teorema de Incompletitud de Gödel, al que nos referiremos más adelante.

Ejemplo 2.1. Lenguaje del conjunto $\mathfrak{L} - (RE \cup CO\text{-}RE)$

Sea $L \in RE - R$, y por lo tanto $L^C \notin RE$. Si $L_{01} = \{0w \mid w \in L\} \cup \{1w \mid w \in L^C\}$, entonces $L_{01} \notin RE \cup CO\text{-}RE$:

- Se cumple que $L_{01} \notin RE$. Supongamos que $L_{01} \in RE$. Entonces existe una MT M_{01} que reconoce L_{01}. A partir de M_{01} se va a construir una MT M que reconoce L^C (absurdo porque $L^C \notin RE$), por lo que M_{01} no puede existir:
  Dada una entrada w, M genera la cadena 1w, luego simula M_{01} a partir de 1w, y responde como M_{01}. De esta manera, $L^C = L(M)$: $w \in L^C \leftrightarrow 1w \in L_{01} \leftrightarrow M_{01}$ acepta $1w \leftrightarrow$ M acepta w.
- Se cumple también que $L_{01} \notin CO\text{-}RE$. Supongamos que $L_{01} \in CO\text{-}RE$, o lo que es lo mismo, que $L_{01}{}^C \in RE$. Entonces existe una MT $M_{01}{}^C$ que reconoce $L_{01}{}^C$. Se va a construir una MT M' que reconoce L^C (absurdo porque $L^C \notin RE$), por lo que $M_{01}{}^C$ no puede existir:
  Dada una entrada w, M' genera la cadena 0w, luego simula $M_{01}{}^C$ a partir de 0w, y responde como $M_{01}{}^C$. De esta manera $L^C = L(M')$: $w \in L^C \leftrightarrow 0w \in L_{01}{}^C \leftrightarrow M_{01}{}^C$ acepta $0w \leftrightarrow$ M' acepta w.

Fin de Ejemplo

Las diferencias en el grado de dificultad de los lenguajes de las cuatro categorías señaladas se perciben a veces mejor cuando se trata con MT restringidas, es decir con MT con restricciones en sus movimientos, sus cintas, etc. Esto se analizará con cierto detalle en la Clase 5. Por ejemplo, dado un conjunto de problemas indecidibles en el marco de las MT generales, cuando se consideran con MT restringidas determinados problemas se mantienen indecidibles mientras que otros pasan a ser decidibles. En el caso de estos últimos problemas, además, se pueden destacar entre ellos diferencias en el costo de resolución (tiempo de ejecución de las MT).

Ejercicios de la Clase 2

1. Completar las pruebas del Teorema 2.1.
2. Probar que la clase R es cerrada con respecto a la unión.
3. Probar que todo lenguaje finito es recursivo.
4. Completar las pruebas del Teorema 2.2.

5. Construir una MT distinta a la del Teorema 2.2 para probar que la clase RE es cerrada con respecto a la unión, tal que se detenga y rechace cuando las dos simulaciones "en paralelo" rechazan.
6. Probar que RE es cerrada con respecto a la intersección.
7. Completar la prueba del Teorema 2.3.
8. Dados dos lenguajes L_1 y L_2, determinar:
   i. Si $L_1 \in RE$, ¿para qué casos se cumple $\Sigma^* - L_1 \in RE$?
   ii. Si $L_1 \in R$ y se le agrega una cantidad finita de cadenas, ¿el lenguaje resultante sigue siendo recursivo?
   iii. Si $L_1 \in RE - R$ y se le quita una cantidad finita de cadenas, ¿el lenguaje resultante sigue perteneciendo a RE – R?
   iv. Si L_1 y $L_2 \in RE$, ¿$L_1 - L_2 \in RE$?
   v. Si L_1 y $L_2 \in$ CO-RE, ¿$L_1 \cap L_2 \in$ CO-RE?
   vi. Si $L_2 \in RE$ y $L_1 \subseteq L_2$, ¿$L_1 \in RE$?
   vii. Si $L_1 \cap L_2 \in RE$, ¿L_1 o $L_2 \in RE$?
   viii. Si $L_1 \cup L_2 \in RE$, ¿L_1 o $L_2 \in RE$?
9. Probar que si $L_1, L_2, \ldots, L_k$, son recursivamente numerables, disjuntos dos a dos, y su unión es Σ^*, entonces son recursivos.
10. Sean L_1 y L_2 dos lenguajes recursivamente numerables de números naturales representados en notación unaria. Probar que también es recursivamente numerable el lenguaje $L = \{x \mid x$ es un número natural representado en notación unaria, y existen y, z, tales que $x = y + z$, con $y \in L_1$ y $z \in L_2\}$.
11. Dada una MT M_1, construir una MT M_2 que establezca si $L(M_1) \neq \emptyset$. ¿Se puede construir además una MT M_3 para establecer si $|L(M_1)| \leq 1$? Justificar la respuesta.
12. Construir una MT que genere todos los textos que pueden escribirse con un alfabeto recursivamente numerable.
13. Probar que $L \in RE$ y $L \neq \emptyset$, si y sólo si existe una MTD M tal que $L = \{y \mid \exists x$, con $M(x) = y\}$, es decir, L es el codominio de la función computada por M.
14. Se define que un lenguaje infinito es recursivamente numerable sin repeticiones, si es el codominio de una función computable inyectiva. Probar que L es recursivamente numerable sin repeticiones, si y sólo si L es infinito y recursivamente numerable.
15. Se define que un lenguaje es recursivamente numerable en orden creciente, si 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, si y sólo si L es infinito y recursivo.

Clase 3. Indecidibilidad

El contenido de esta clase, y también el de la siguiente, se centra en los problemas indecidibles, es decir en los problemas de decisión asociados a los lenguajes no recursivos, los lenguajes de Ω – R. Ya nos vamos a ocupar de los problemas decidibles en la segunda parte del libro, cuando analicemos su complejidad computacional temporal.

Inicialmente presentamos un primer lenguaje recursivamente numerable que no es recursivo, que completa la formalización de la jerarquía de la computabilidad definida en la clase anterior. Para ello nos valemos de la técnica de *diagonalización*. Mostramos luego distintos ejemplos de aplicación de esta técnica, y a partir de algunos de ellos obtenemos otros lenguajes no recursivos.

En la siguiente clase encontraremos más representantes del conjunto Ω – R pero con otra técnica, la *reducción de problemas*. Habiendo logrado probar la existencia de algunos primeros lenguajes de esta naturaleza con cierto esfuerzo usando diagonalización, luego la obtención a partir de los mismos de nuevos representantes será más sencilla empleando reducciones de problemas.

Para los desarrollos de esta clase tenemos que introducir antes que nada el concepto de *máquina de Turing universal*, el cual utilizaremos sobremanera en las diagonalizaciones y reducciones. Hasta ahora hemos trabajado con MT que tratan, cada una, un problema particular. Pero como modelo adecuado de una computadora la máquina de Turing puede ser también "programable", capaz de ejecutar cualquier MT M a partir de cualquier entrada w. En este caso, identificando con U a la MT universal, U cumple que:

- Sus entradas son pares de la forma (<M>, w), siendo <M> la codificación de una MT M, y w una entrada, también codificada, de M.
- Dada la entrada (<M>, w), su ejecución consiste en simular M a partir de w.

La MT universal U es como toda MT una tupla $(Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$, con sus propios estados y símbolos, y una función de transición δ que establece cómo simular la MT cuyo código forma parte de su entrada. Una posible manera de codificar una MT es la

siguiente (para simplificar se consideran MT con una sola cinta; la codificación se puede generalizar fácilmente a varias cintas). Dada $M = (Q, \Sigma, \Gamma, \delta, q_1, q_A, q_R)$:

- Los estados del conjunto $Q = \{q_1, \ldots, q_k\}$ se representan por los números 1, …, k, en notación binaria. Además, q_A y q_R se representan en binario por los números k + 1 y k + 2, respectivamente.
- Los símbolos del alfabeto $\Gamma = \{x_{i1}, \ldots, x_{in}\}$ se representan por los números i_1, …, i_n, en notación binaria.
- Los movimientos L, R y S se representan, respectivamente, por los números 1, 2 y 3, en notación binaria.
- Finalmente, la función de transición δ se representa por una secuencia de 5-tuplas. Las tuplas se separan por el símbolo numeral, es decir #, y sus componentes por medio de una coma.

Con estas consideraciones, la codificación de una MT M consiste en la cadena formada por |Q|, en notación binaria, seguida por el separador # y luego por la representación de la función de transición δ. Se antepone |Q| para que se puedan identificar los estados q_A y q_R. Por ejemplo, si $M = (\{q_1\}, \{x_4, x_5\}, \{x_1, x_4, x_5\}, \delta, q_1, q_A, q_R)$, y la función de transición se define con $\delta(q_1, x_4) = (q_A, x_1, S)$ y $\delta(q_1, x_5) = (q_R, x_1, R)$, entonces el código de la MT M es

$$\langle M \rangle = 1\#1,100,10,1,11\#1,101,11,1,10$$

La cadena w en (<M>, w) en realidad debe entenderse como la representación de w, con sus símbolos representados en notación binaria y separados entre sí por una coma. Para separar <M> de w se utilizan dos numerales consecutivos.

Codificar las MT permite que las mismas se puedan enumerar, por ejemplo utilizando el orden canónico. La siguiente MT computa, dado un número natural i, el código de la MT i-ésima según el orden canónico:

1. Hace n := 0.
2. Crea la siguiente cadena w según el orden canónico.
3. Verifica si w es un código válido de MT. Si no, vuelve al paso 2.
4. Si i = n, escribe w en la cinta de salida y se detiene en q_A.

5. Hace n := n + 1.
6. Vuelve al paso 2.

En el paso 2 se generan cadenas en orden canónico con los símbolos que pueden formar parte del código de una MT, a partir de un ordenamiento establecido entre los símbolos. En el paso 3 se verifica que las cadenas generadas tengan la forma |Q|#<δ>, que los componentes |Q| y <δ> respeten la codificación establecida, que ningún código de estado en <δ> supere el valor |Q| + 2, etc.
Cuando en el paso 4 se cumple la igualdad i = n, significa que se obtuvo el código $<M_i>$.
Notar que en la enumeración canónica obtenida de las MT M_0, M_1, M_2, …, puede darse, para índices distintos i, k, que $L(M_i) = L(M_k)$.
Con estos elementos preparatorios ya estamos en condiciones de probar la existencia de un primer lenguaje de RE – R.

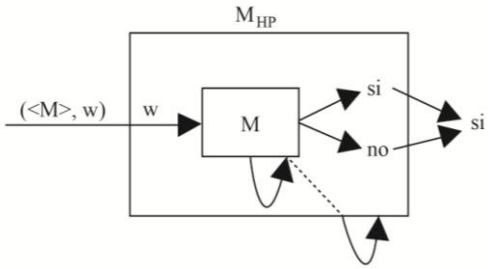
Teorema 3.1. El problema de la detención es computable no decidible

El problema de la detención (de las MT), también conocido como HP (por *halting problem*), es un problema clásico de la literatura, que se enuncia de la siguiente manera: dada una MT M y una cadena w, ¿M se detiene a partir de w?
El lenguaje que lo representa es HP = {(<M>, w) | la MT M se detiene a partir de w}.
Probamos a continuación que HP ∈ RE – R; primero que HP ∈ RE, y luego que HP ∉ R.

Prueba de que HP ∈ RE.
Sea la siguiente MT M_{HP}, que a partir de una entrada v, hace:

1. Si v no es un código válido (<M>, w), rechaza.
2. Simula M a partir de w. Si M se detiene, acepta.

La figura siguiente ilustra el comportamiento de M_{HP}:



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

Prueba de que $HP \notin R$.

Supongamos que la siguiente tabla infinita T de unos y ceros describe el comportamiento de todas las MT M_0, M_1, …, con respecto a todas las cadenas de Σ^* w_0, w_1, …, en lo que hace a la detención, considerando el orden canónico. El valor de $T[M_i, w_k]$ es 1 o 0, según la MT M_i se detiene o no se detiene a partir de la cadena w_k, respectivamente:

	w_0	w_1	w_2	w_3	…	w_m	…	w_h	w_{h+1}	…
M_0	1	0	1	1	…	1	…	1	0	…
M_1	1	0	0	1	…	0	…	0	0	…
M_2	0	0	1	0	…	1	…	0	0	…
M_3	0	1	1	1	…	1	…	1	1	…
…	…	…	…	…	…	…	…	…	…	…
M_m	1	1	0	1	…	0	…	1	1	…
…	…	…	…	…	…	…	…	…	…	…
M_h	0	0	0	1	…	0	…	1	0	…
M_{h+1}	0	0	1	0	…	0	…	1	1	…
…	…	…	…	…	…	…	…	…	…	…

Sea el lenguaje $D = \{w_i \mid M_i$ se detiene a partir de $w_i\}$. Es decir, los elementos de D son las cadenas w_i tales que $T[M_i, w_i] = 1$. Vamos a probar que si $HP \in R$ entonces $D \in R$, y que $D \notin R$, por lo que se cumplirá que $HP \notin R$.

Prueba de que $HP \in R \rightarrow D \in R$.

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). Más precisamente, si HP ∈ R, entonces existe una MT M_{HP} que se detiene siempre y reconoce HP. Para probar que D ∈ R, se va a construir una MT M_D que se detiene siempre y reconoce D. A partir de una entrada w, M_D trabaja de la siguiente manera:

1. Encuentra el índice i tal que $w = w_i$, según el orden canónico.
2. Genera <M_i>.
3. Simula M_{HP} a partir de (<M_i>, w_i), y acepta (respectivamente rechaza) si M_{HP} acepta (respectivamente rechaza).

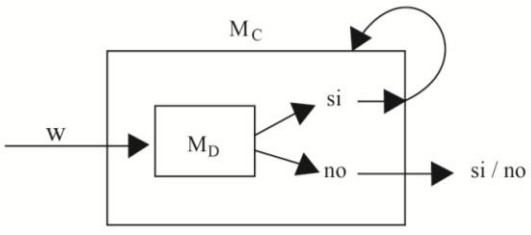
En el paso 1, la MT M_D genera canónicamente todas las cadenas hasta que encuentra la i-ésima. Como M_{HP} se detiene siempre, entonces M_D se detiene siempre. Además se cumple que $D = L(M_D)$: $w_i \in D \leftrightarrow M_i$ se detiene a partir de $w_i \leftrightarrow M_{HP}$ acepta (<M_i>, w_i) $\leftrightarrow M_D$ acepta w_i.

Prueba de que $D \notin R$.

Supongamos que D ∈ R. Sea M_D una MT que se detiene siempre y reconoce el lenguaje D. A partir de M_D construimos la siguiente MT M_C. Dada una entrada w, M_C hace:

1. Simula M_D a partir de w.
2. Si M_D acepta, M_C entra en un *loop*. Y si M_D rechaza, M_C acepta (en realidad da lo mismo si rechaza).

La figura siguiente ilustra el comportamiento de la MT M_C:



La definición del fragmento de la función de transición de M_C correspondiente al *loop* no tiene mayores complicaciones: por ejemplo, se puede reemplazar q_A por un estado nuevo q y definir una 5-tupla (q, x, q, x, S) por cada símbolo x del alfabeto. En el paso 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, en lo que respecta al comportamiento de la MT M_i a partir de la entrada w_i: si M_D establece que M_i se detiene a partir de w_i (es decir si M_D acepta), entonces M_C no se detiene a partir de w_i, y si M_D establece que M_i no se detiene a partir de w_i (es decir si M_D rechaza), entonces M_C se detiene a partir de w_i. Como la diagonal de la tabla T representa el comportamiento de las MT M_i respecto de las cadenas w_i, entonces "el complemento a dos" de la diagonal (se permutan unos por ceros y ceros por unos), que difiere de todas las filas de T, representa el comportamiento de M_C respecto de las cadenas w_i. Esto entonces va a implicar que M_C no es ninguna de las MT M_i, lo que se formaliza a continuación. Dado que la tabla T incluye a todas las MT, M_C tiene que ser alguna de ellas, digamos M_m:

	w_0	w_1	w_2	w_3	...	w_m	...	w_h	w_{h+1}	...
M_0	1	0	1	1	...	1	...	1	0	...
M_1	1	0	0	1	...	0	...	0	0	...
M_2	0	0	1	0	...	1	...	0	0	...
M_3	0	1	1	1	...	1	...	1	1	...
...	...	...	...	...	...	...	...	...	...	...
$M_{m\,=}\,M_C$	1	1	0	1	...	0	...	1	1	...
...	...	...	...	...	...	...	...	...	...	...
M_h	0	0	0	1	...	0	...	1	0	...
M_{h+1}	0	0	1	0	...	0	...	1	1	...
...	...	...	...	...	**...**	...	...	...	...	**...**

Veamos qué sucede considerando la MT M_D y la entrada w_m:

a. Si M_D acepta w_m, entonces por cómo se construyó M_C se cumple que M_C no se detiene a partir de w_m. Pero $M_C = M_m$, así que vale que M_m no se detiene a partir de w_m. Por lo tanto, por la definición de M_D se cumple que M_D rechaza w_m, lo que contradice la hipótesis.
b. Si en cambio M_D rechaza w_m, entonces por cómo se construyó M_C se cumple que M_C se detiene a partir de w_m. Pero como $M_C = M_m$ entonces M_m se detiene a partir de w_m. Dada la definición de M_D queda que M_D acepta w_m, lo que otra vez contradice la hipótesis.

De esta manera no puede existir la MT M_C. O dicho en otras palabras, M_C es distinta de todas las MT enumeradas en T, lo que es absurdo porque en T están todas las MT. Y como la MT M_C se construyó a partir de la MT M_D, entonces tampoco puede existir M_D. Esto significa que $D \notin R$.

Fin de Teorema

La técnica empleada en el teorema anterior se denomina diagonalización. Es muy útil para probar que dos conjuntos difieren en al menos un elemento, al que se lo suele denominar *separador*. En el caso anterior, el separador que encontramos entre las clases RE y R es el lenguaje HP (en realidad también el lenguaje D). La idea de la diagonalización es muy sencilla. En una de sus variantes, se toma como base una tabla de unos y ceros y se utiliza el hecho de que "el complemento a dos" de su diagonal difiere de todas las filas: difiere de la primera fila en el primer elemento, de la segunda fila en el segundo elemento, y así sucesivamente.

Por ejemplo, con las dos diagonalizaciones que se presentan a continuación, probamos que los números reales y el conjunto de partes de los números naturales no son numerables (en la Clase 5 identificamos formalmente a los lenguajes recursivamente numerables con los que se pueden enumerar).

Ejemplo 3.1. El conjunto de los números reales no es numerable

G. Cantor demostró por diagonalización (una variante de la forma vista previamente) que hay más números reales que números naturales, es decir $|R| > |N|$, o lo que es lo mismo, que el conjunto de números reales no es numerable. La prueba es la siguiente. Se parte de una posible enumeración de los números reales entre 0 y 1 (se puede hacer lo mismo para el conjunto completo R de los números reales, ya que existe una biyección entre R y el intervalo real (0, 1)):

$$0 \rightarrow 0{,}1035762718\ldots$$
$$1 \rightarrow 0{,}1432980611\ldots$$
$$2 \rightarrow 0{,}0216609521\ldots$$
$$3 \rightarrow 0{,}4300535777\ldots$$
$$4 \rightarrow 0{,}9255048910\ldots$$
$$5 \rightarrow 0{,}5921034329\ldots$$
$$6 \rightarrow 0{,}6366791045\ldots$$

$7 \rightarrow 0{,}8705007419\ldots$

$8 \rightarrow 0{,}0431173780\ldots$

$9 \rightarrow 0{,}4091673889\ldots$

…….

Considerando la diagonal de la tabla de dígitos definida por los decimales de los números reales enumerados, es decir la diagonal formada por el primer decimal del número real con índice 0, el segundo decimal del número real con índice 1, el tercer decimal del número real con índice 2, etc., queda la siguiente secuencia:

1, 4, 1, 0, 0, 3, 1, 4, 8, 9, …

Modificando la secuencia de manera tal que cuando hay un decimal 1 se lo reemplaza por el decimal 2, y cuando hay un decimal distinto de 1 se lo reemplaza por el decimal 1, queda

2, 1, 2, 1, 1, 1, 2, 1, 1, 1, …

De esta manera, el número real 0,2121112111… difiere de todos los números reales de la enumeración: difiere del primero en su primer decimal, del segundo en su segundo decimal, del tercero en su tercer decimal, y así sucesivamente. Esto es absurdo porque partimos de la base de una enumeración de todos los números reales entre 0 y 1. Por lo tanto, los números reales entre 0 y 1 no son numerables.

Fin de Ejemplo

Ejemplo 3.2. El conjunto de partes de los números naturales no es numerable

Sea X un conjunto numerable de conjuntos de números naturales. Se prueba por diagonalización que $X \neq P(N)$, siendo N el conjunto de los números naturales y P(N) el conjunto de partes de N. Por lo tanto esto demuestra que P(N) no es numerable. Llamando X_n a los conjuntos de X, con $n \geq 0$, sea $C = \{n \in N \mid n \notin X_n\}$. Se cumple:

- $C \subseteq N$, por definición.

- $C \neq X_n$ para todo n:

  $C \neq X_0$ porque difieren en el número 0: $0 \in C \leftrightarrow 0 \notin X_0$

  $C \neq X_1$ porque difieren en el número 1: $1 \in C \leftrightarrow 1 \notin X_1$

  $C \neq X_2$ porque difieren en el número 2: $2 \in C \leftrightarrow 2 \notin X_2$

  y así sucesivamente.

Por lo tanto, encontramos un elemento C de P(N) que no está en X. Como partimos de un X arbitrario, hemos demostrado que P(N) no es numerable. Dada cualquier enumeración de sus elementos, siempre se puede encontrar un conjunto de números naturales que no está en la enumeración. Notar que si T es una tabla de unos y ceros, con T[n, i] = 1 si $i \in X_n$ y T[n, i] = 0 si $i \notin X_n$, entonces la fila n de T representa el conjunto X_n, y "el complemento a dos" de la diagonal de T representa el conjunto C, lo que explica por qué C difiere de todos los conjuntos X_n.

Fin de Ejemplo

Retomando los ejemplos con MT, en la última diagonalización de esta clase que presentamos a continuación, se prueba de una manera alternativa a la presentada previamente que $RE \subset \mathfrak{L}$.

Ejemplo 3.3. Otro lenguaje que no es recursivamente numerable

Se puede probar por diagonalización que $RE \subset \mathfrak{L}$ de una manera muy similar a cómo se probó que $HP \notin R$. En la siguiente tabla infinita S de unos y ceros, $S[M_i, w_k] = 1$ o 0 según la MT M_i acepta o rechaza la cadena w_k, respectivamente:

	w_0	w_1	w_2	w_3	…	w_m	…	w_h	w_{h+1}	…
M_0	1	0	1	1	…	1	…	1	0	…
M_1	1	0	0	1	…	0	…	0	0	…
M_2	0	0	1	0	…	1	…	0	0	…
M_3	0	1	1	1	…	1	…	1	1	…
…	…	…	…	…	…	…	…	…	…	…
M_m	1	1	0	1	…	0	…	1	1	…
…	…	…	…	…	…	…	…	…	…	…
M_h	0	0	0	1	…	0	…	1	0	…
M_{h+1}	0	0	1	0	…	0	…	1	1	…
…	…	…	…	…	…	…	…	…	…	…

De esta manera, $L(M_i) = \{w_k \mid S[M_i, w_k] = 1\} = \{w_k \mid M_i \text{ acepta } w_k\}$. Si $L_i = L(M_i)$, y $E = \{w_i \mid S[M_i, w_i] = 1\} = \{w_i \mid M_i \text{ acepta } w_i\}$, se cumple que $E^C \notin RE$:

- $E^C \neq L_0$ porque $w_0 \in E^C \leftrightarrow w_0 \notin L_0$
- $E^C \neq L_1$ porque $w_1 \in E^C \leftrightarrow w_1 \notin L_1$
- $E^C \neq L_2$ porque $w_2 \in E^C \leftrightarrow w_2 \notin L_2$

  y así sucesivamente.

Es decir, E^C no es ninguno de los lenguajes L_i , y como los lenguajes L_i son todos los lenguajes recursivamente numerables dado que en la tabla S están todas las MT, entonces se cumple que $E^C \notin RE$. En términos de la tabla S, "el complemento a dos" de su diagonal, que representa el lenguaje E^C, difiere de todas las filas de S, que representan todos los lenguajes recursivamente numerables.

Fin de Ejemplo

Notar que con lo que hemos desarrollado hasta este momento contamos con distintos caminos para formalizar los límites de la computabilidad, es decir para probar la inclusión estricta $RE \subset \mathfrak{L}$. Por ejemplo, $HP^C \notin RE$, porque de lo contrario HP sería recursivo. Otro camino es recurrir directamente al lenguaje artificial L_{01} que presentamos en la clase anterior, que no pertenece a RE (ni siquiera pertenece a CO-RE). Una tercera alternativa es la diagonalización empleada en el ejemplo anterior, con la que se encontró el lenguaje separador E^C entre $\mathfrak{L}$ y RE. Hay aún otra manera de probar que no todos los lenguajes son recursivamente numerables, basada en la cardinalidad de los conjuntos infinitos: no puede haber más MT, y así lenguajes recursivamente numerables, que $|\Sigma^*|$; la cantidad de lenguajes de $\mathfrak{L}$ es $|P(\Sigma^*)|$; y como $|\Sigma^*| < |P(\Sigma^*)|$, entonces $RE \subset \mathfrak{L}$.

El Ejemplo 3.3 aporta también un camino alternativo al del teorema del problema de la detención para probar que $R \subset RE$. Claramente, el lenguaje $E = \{w_i \mid M_i \text{ acepta } w_i\}$ es recursivamente numerable, y como vimos que E^C no es recursivamente numerable, entonces se cumple que E no es recursivo.

Otro lenguaje clásico de RE – R es $L_U = \{(<M_i>, w_k) \mid M_i \text{ acepta } w_k\}$. Claramente, el lenguaje L_U es recursivamente numerable, y no es recursivo porque de lo contrario el

lenguaje E sería recursivo (la prueba queda como ejercicio). L_U representa el problema de la pertenencia de una cadena a un lenguaje, o directamente el problema de la pertenencia. Se lo conoce como *lenguaje universal*. En la clase siguiente lo utilizaremos muy a menudo para aplicar la técnica de reducción de problemas.

Concluimos esta clase con una apreciación sobre el significado de la indecidibilidad del problema de la detención, generalizándola a todos los problemas indecidibles. Hemos demostrado la insolubilidad algorítmica del problema *general* de la detención, es decir, hemos probado que no existe ninguna MT que lo resuelve *sistemáticamente*, considerando las infinitas instancias ($<M_i>$, w_k). De todos modos ciertamente existe un algoritmo que resuelve el problema cuando se considera una instancia *particular* ($<M_i>$, w_k): uno que acepta o uno que rechaza. Para la elección del algoritmo adecuado en cada caso habrá que apelar al ingenio, sin una técnica estándar en principio. En otras palabras, un problema con una sola instancia siempre es decidible (obviamente nuestro interés radica en los problemas con infinitas instancias).

Otro ejemplo que ilustra esta dualidad de lo general y lo particular es el problema, ya mencionado, relacionado con las ecuaciones diofánticas: dada una ecuación algebraica con distintas variables y coeficientes enteros, ¿existe una solución con números enteros? Por ejemplo, dada la ecuación $xy^2 + 5x^3 + 8xz = 0$, ¿existen valores enteros de x, y, z, que la resuelven? Se prueba que este problema es indecidible. Por otro lado, ecuaciones diofánticas particulares son las que consideró P. de Fermat cuando planteó su famoso teorema, conocido como Ultimo Teorema de Fermat, en el siglo XVII: dado $n > 2$, la ecuación $x^n + y^n = z^n$, siendo x e y mayores que 0, no tiene solución entera (Fermat no mostró la prueba del teorema justificándose por lo pequeño del margen del libro en que lo publicó; el teorema se demostró recién en 1995). El problema de Fermat tiene una sola instancia, y como tal es decidible.

Un último ejemplo de este tipo es el de la Conjetura de Goldbach, que establece que todo número natural par mayor que 2 puede expresarse como la suma de dos números primos. Por tener una sola instancia, este problema es decidible (al día de hoy sigue sin demostrarse). Naturalmente, y al igual que el problema de Fermat, la conjetura de Goldbach se probaría fácilmente si el problema de la detención fuese decidible.

Ejercicios de la Clase 3

1. Proponer cómo codificar una MT con varias cintas.
2. 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) o x = (<M>, w) y M no acepta w

Determinar si f es una función total computable.

3. Probar que si el lenguaje $L_U = \{(<M_i>, w_k) \mid M_i \text{ acepta } w_k\}$ fuera recursivo, entonces también el lenguaje $E = \{w_i \mid M_i \text{ acepta } w_i\}$ sería recursivo.
4. Construir una MT que genere todos los índices i tales que M_i acepta w_i considerando el orden canónico.
5. Construir una MT que genere los códigos de todas las MT que a partir de la cadena vacía λ se detienen en a lo sumo 100 pasos.
6. Probar que los siguientes lenguajes son recursivos:
   i. L = {<M> | la MT M, a partir de la cadena vacía λ, escribe alguna vez un símbolo no blanco}.
   ii. L = {(<M>, w) | la MT M, a partir de w, nunca lee una celda más de una vez}.
   iii. L = {(<M>, w) | la MT M, a partir de w, nunca se mueve a la izquierda}.
7. Plantear una manera de enumerar los siguientes conjuntos de números:
   i. Los números enteros pares.
   ii. Los números racionales.
8. Asumiendo que el problema de la detención (de las MT) es decidible, construir una MT que resuelva los siguientes problemas:
   i. El Ultimo Teorema de Fermat.
   ii. La Conjetura de Goldbach.
9. Se define que dos lenguajes disjuntos L_1 y L_2 son *recursivamente inseparables* si no existe ningún lenguaje recursivo A tal que $L_1 \cap A = \emptyset$ y $L_2 \subset A$. Por ejemplo, HP y HP^C son claramente recursivamente inseparables. Otro caso es el del par de lenguajes L_1 = {<M> | M a partir de <M> se detiene y acepta} y L_2 = {<M> | M a partir de <M> se detiene y rechaza}, que se prueba de la siguiente manera. Supongamos que existe un lenguaje recursivo A que separa L_1 y L_2, con $L_1 \cap A = \emptyset$ y $L_2 \subset A$ (para el caso $L_2 \cap A = \emptyset$ y $L_1 \subset A$ la prueba se hace con A^C). Sea M una MT que acepta A y se detiene siempre. Si M a partir de <M> se detiene y acepta, entonces <M> ∈ L_1, y por lo tanto <M> ∉ A, por lo que M a partir de <M> se detiene y rechaza (absurdo). Se llega también a un absurdo suponiendo que M a partir de <M> se detiene y rechaza.

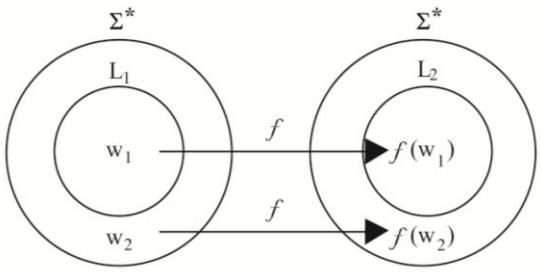
Se pide probar que también son recursivamente inseparables los lenguajes L_1 = {<M> | M a partir de la cadena vacía λ se detiene y acepta} y L_2 = {<M> | M a partir de la cadena vacía λ se detiene y rechaza}.

Clase 4. Reducciones de problemas

En esta clase completamos nuestro análisis de los problemas indecidibles, ahora con foco en la técnica de reducción de problemas, que nos permitirá encontrar lenguajes no recursivos y no recursivamente numerables de una manera en general mucho más sencilla que por medio de la diagonalización. Consideramos en su gran mayoría problemas relacionados con MT. Al final, a partir de un teorema probado con una reducción de problemas, presentamos una técnica que facilita aún más la demostración de la no recursividad, para un determinado tipo de lenguajes. La noción de reducción de problemas es muy simple: para resolver un problema se lo relaciona con otro, que se sabe cómo resolverlo; a partir de este conocimiento se resuelve el problema original.

Definición 4.1. Reducción de problemas

Sean L_1 y L_2 dos lenguajes incluidos en Σ^*. Existe una reducción del lenguaje L_1 al lenguaje L_2, si y sólo si 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*. Que f sea total computable significa, como se indicó previamente, que existe una MT que a partir de cualquier cadena w computa f(w) en su cinta de salida y se detiene. En general, identificaremos con M_f a la MT que computa f. La figura siguiente 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 toda cadena de L_1 en una cadena de L_2, y toda cadena no perteneciente a L_1 en una cadena no perteneciente a L_2.



Utilizaremos la notación L_1 α L_2 para expresar que existe una reducción del lenguaje (o problema) L_1 al lenguaje (o problema) L_2.

Fin de Definición

Ejemplo 4.1. Reducción de L_{200} a L_{100}

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} α L_{100}.

Definición de la función de reducción.

Sea la función de reducción f: $\Sigma^* \rightarrow \Sigma^*$, con

$$f((x, y, z)) = (x^2, y^2, z^2)$$

cuando la entrada es válida sintácticamente. En caso contrario f genera una salida inválida, por ejemplo la cadena 1.

La función f es total computable.

Se puede construir fácilmente una MT M_f que chequea, dada una entrada, si tiene o no la forma (x, y, z), y que luego genera, correspondientemente, la terna (x^2, y^2, z^2) o la cadena 1.

Se cumple $(x, y, z) \in L_{200} \leftrightarrow f(x, y, z) \in L_{100}$.

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

Fin de Ejemplo

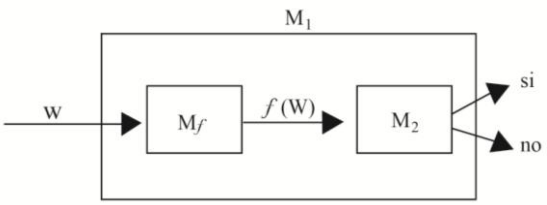
Tal como lo refleja el ejemplo anterior, para probar que L_1 α L_2 hay que probar que existe una MT M_f que computa una función total f, y que $w \in L_1 \leftrightarrow f(w) \in L_2$ para toda cadena de Σ^*. Notar la utilidad de la reducción anterior. Si se tiene una MT M que reconoce el lenguaje L_{100}, entonces también se tiene una MT que reconoce el lenguaje L_{200}, componiendo M_f con M: si M acepta o rechaza (x^2, y^2, z^2) significa que (x, y, z) pertenece o no pertenece, respectivamente, a L_{200}. Esto se formaliza en el siguiente teorema.

Teorema 4.1. Si L_2 está en R (RE) y L_1 α L_2, entonces L_1 está en R (RE)

Probaremos que si $L_2 \in R$ y L_1 α L_2, entonces $L_1 \in R$. La prueba de que si $L_2 \in RE$ y L_1 α L_2, entonces $L_1 \in RE$, es muy similar y queda como ejercicio.

Idea general.

Componiendo la MT M_f que computa la función de reducción f del lenguaje L_1 al lenguaje L_2, con la MT M_2 que reconoce el lenguaje L_2 y se detiene siempre, se obtiene una MT M_1 que reconoce el lenguaje L_1 y se detiene siempre (ver la figura siguiente):



Construcción de la MT M_1.

Sea M_f una MT que computa la función de reducción f, con $w \in L_1 \leftrightarrow f(w) \in L_2$, y sea M_2 una MT que reconoce L_2 y se detiene siempre. La MT M_1 trabaja de la siguiente manera:

1. Simula M_f a partir de la entrada w y obtiene f(w).
2. Simula M_2 a partir de f(w) y acepta si y sólo si M_2 acepta.

Prueba de que M_1 se detiene siempre.

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

Prueba de $L_1 = L(M_1)$.

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

Fin de Teorema

Como corolario del teorema anterior se establece que si L_1 no es recursivo y existe una reducción de L_1 a L_2, entonces L_2 tampoco es recursivo (de lo contrario L_1 sería recursivo). Lo mismo se puede decir para el caso de los lenguajes recursivamente numerables.

Por lo tanto, las reducciones se pueden emplear también para probar que un lenguaje no es recursivo o no es recursivamente numerable (en realidad, se pueden aplicar sobre una gama mucho mayor de clases de lenguajes, como se verá en la segunda parte del libro). Más aún, en lo que sigue nos enfocaremos en esta segunda visión, para desarrollar pruebas "negativas", es decir pruebas de no pertenencia (se aplicarán con el mismo objetivo que las diagonalizaciones). Para las pruebas "positivas", las de pertenencia, el camino seguirá siendo la construcción de MT.

Se presentan a continuación varios ejemplos de reducciones de problemas. Además de la ejercitación en el uso de la técnica, se irán poblando las distintas clases de lenguajes de la jerarquía de la computabilidad. Se tratan en su gran mayoría problemas relacionados con MT.

Ejemplo 4.2. Reducción de HP a L_U

Se probó por diagonalización en la clase anterior (Teorema 3.1) que el lenguaje recursivamente numerable HP = {(<M>, w) | M se detiene a partir de w}, que representa el problema de la detención, no es recursivo.

También a partir de una diagonalización se estableció la no recursividad de otro lenguaje clásico, recursivamente numerable, de la computabilidad, el lenguaje L_U o lenguaje universal, definido por L_U = {(<M>, w) | M acepta w}, que representa el problema de la pertenencia.

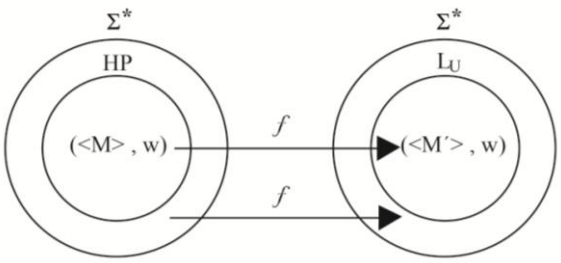
Una manera alternativa de probar que L_U no es recursivo, aplicando el corolario del teorema anterior, es construir directamente una reducción de HP a L_U.

Definición de la función de reducción.

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

$$f((<M>, w)) = (<M'>, w)$$

tal que M' se comporta como M, salvo que cuando M se detiene, ya sea en el estado q_A o en el estado q_R, M' acepta (ver la figura siguiente):



La función f es total computable.

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

Se cumple (<M>, w) ∈ HP ↔ f((<M>, w)) ∈ L_U.

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

Fin de Ejemplo

De esta manera, para probar que L_U no es recursivo, se construyó una reducción desde un lenguaje no recursivo, HP, a L_U. L_U no puede ser recursivo, porque si lo fuera también lo sería HP (absurdo): combinando M_f con una supuesta MT que acepta L_U y se detiene siempre, se obtendría una MT que acepta HP y se detiene siempre. Planteándolo 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.

No había muchas opciones de lenguajes de los cuales reducir a L_U. Con un conjunto más rico de alternativas, que se irán presentando en los ejemplos siguientes, la elección del lenguaje conocido suele basarse en determinadas características del mismo y del lenguaje nuevo, que facilitan la construcción de la MT M_f.

Notar que si no se hubiera tenido indicios acerca de la no recursividad del lenguaje L_U, una primera aproximación para determinar la ubicación de L_U en la jerarquía de la computabilidad podría haber sido la siguiente:

- Construir una MT que a partir de (<M>, w) simula M a partir de w.
- Como M puede no detenerse a partir de w, entonces establecer 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" de un determinado lenguaje L (la construcción de una MT que pruebe que $L \in R$), o "negativa" (una reducción que pruebe que $L \notin R$). Esta aproximación se presenta en algunos ejemplos más adelante. En el ejemplo siguiente se prueba que también existe una reducción de L_U a HP.

Ejemplo 4.3. Reducción de L_U a HP

Definición de la función de reducción.

Para pares válidos (<M>, w), se define

$$f((<M>, w)) = (<M'>, w)$$

tal que M' se comporta como M, salvo que cuando M se detiene en q_R, M' no se detiene.

La función f es total computable.

Si la entrada no es una cadena válida (<M>, w), la MT M_f genera la cadena 1. En caso contrario, para generar <M'> la MT M_f modifica <M> de modo tal que M' entre en un *loop* cuando M se detiene en q_R: por ejemplo, como ya se vio en el Teorema 3.1, se puede reemplazar q_R por un estado nuevo q y definir una 5-tupla (q, x, q, x, S) por cada símbolo x del alfabeto de M.

Se cumple $(<M>, w) \in L_U \leftrightarrow f((<M>, w)) \in HP$.

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

Fin de Ejemplo

Se pueden plantear numerosas reducciones útiles considerando HP y L_U. En la clase anterior, en la prueba de que el lenguaje HP no es recursivo (Teorema 3.1), se utilizó un lenguaje similar $D = \{w_i \mid M_i \text{ se detiene a partir de } w_i\}$. Se demostró primero que $HP \in R \rightarrow D \in R$. Notar que esta prueba no es sino una reducción de D a HP. Lo mismo sucedió con L_U en relación al lenguaje similar E. Otras reducciones útiles que se pueden plantear incluyen la visión de MT calculadora. Por ejemplo, se puede reducir HP al

lenguaje de las ternas (<M>, w, v), tales que M(w) = v (es decir que M a partir de w genera v). También se puede reducir L_U al lenguaje de los pares (<M>, w) tales que para cada uno existe una cadena v que cumple M(w) = v. Queda como ejercicio construir dichas reducciones. También los lenguajes HP^C y $L_U{}^C$ son muy útiles para probar mediante reducciones de problemas la no pertenencia a las clases R y RE de numerosos lenguajes.

HP y L_U están entre los lenguajes más difíciles de la clase RE, en el sentido de la computabilidad. Más precisamente, se prueba que todos los lenguajes recursivamente numerables se reducen a ellos (y así, si HP o L_U fueran recursivos, se cumpliría R = RE). Utilizando una terminología más propia de la complejidad computacional, HP y L_U son lenguajes *RE-completos*. Veámoslo para el caso de L_U (la prueba es similar para el caso de HP y queda como ejercicio). Si $L \in RE$ y M es una MT que reconoce L, entonces la función f definida de la siguiente manera:

$$f(w) = (\langle M \rangle, w)$$

para toda cadena w, es claramente una reducción de L a L_U.

En cambio, en la clase R se cumple que, sin considerar los lenguajes Σ^* y $\varnothing$, cualquier lenguaje recursivo L_1 se puede reducir a cualquier lenguaje recursivo L_2, es decir que, siguiendo con la terminología anterior, todo lenguaje recursivo es *R-completo*. Si L_1 y L_2 son dos lenguajes recursivos distintos de Σ^* y $\varnothing$, $a \in L_2$, $b \notin L_2$, y M_1 y M_2 se detienen siempre y reconocen L_1 y L_2, respectivamente, entonces la función f definida de la siguiente manera:

$$f(w) = a, \text{ si } w \in L_1$$
$$f(w) = b, \text{ si } w \notin L_1$$

para toda cadena w, es claramente una reducción de L_1 a L_2.

Antes de seguir con los ejemplos de reducciones de problemas, es útil destacar a esta altura que las reducciones poseen las propiedades de *reflexividad* y *transitividad* (se prueba fácilmente y queda como ejercicio), y que en cambio no son *simétricas*, porque por lo visto recién, todo lenguaje recursivo L se reduce a L_U, y en cambio no puede existir una reducción de L_U a L, porque de lo contrario L_U sería recursivo. Por lo tanto, el hecho de que haya una reducción de L_1 a L_2 no implica que haya una reducción en el

sentido contrario. Para el caso de HP y L_U vimos que se cumple, pero no es lo que sucede en general. Otra propiedad útil para considerar es que existe una reducción de L_1 a L_2 si y sólo si existe una reducción de L_1^C a L_2^C (es la misma función de reducción).

Ejemplo 4.4. Reducción de L_U a L_{Σ^*}

Sea el problema: dada una MT M, ¿M acepta todas las cadenas de Σ*? Se quiere determinar si dicho problema es decidible. El lenguaje que lo representa es $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\}$. Por lo tanto, hay que determinar si L_{Σ^*} es un lenguaje recursivo.

Intuitivamente, no parece ni siquiera que L_{Σ^*} sea recursivamente numerable. Si $L_{\Sigma^*} \in$ RE, entonces existe una MT M_{Σ^*} que se detiene en q_A si y sólo si su entrada <M> es tal que $L(M) = \Sigma^*$. Pero entonces M_{Σ^*} debe aceptar un código <M> después de comprobar que la MT M reconoce las infinitas cadenas de Σ*, lo que no parece razonable. Naturalmente esto no es una demostración formal, pero permite orientar la prueba hacia la búsqueda de una reducción para demostrar que L_{Σ^*} no es recursivo.

Se hará L_U α L_{Σ^*}. Como $L_U \notin R$, entonces $L_{\Sigma^*} \notin R$ (si $L_{\Sigma^*} \in R$, entonces $L_U \in R$). Así se probará que el problema de determinar si una MT acepta todas las cadenas de Σ* es indecidible.

Definición de la función de reducción.

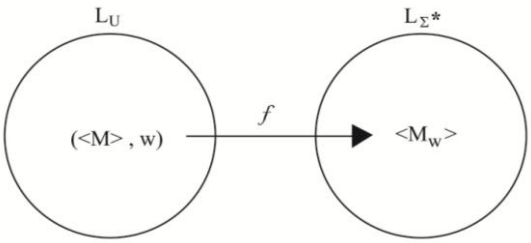
Para pares válidos (<M>, w) se define

$$f((<M>, w)) = <M_w>$$

donde M_w es una MT que borra su entrada v, la reemplaza por w, simula M a partir de w, y acepta si y sólo si M acepta. Se comprueba fácilmente que:

- Si M acepta w, entonces $L(M_w) = \Sigma^*$
- Si M no acepta w, entonces $L(M_w) = \varnothing$ (es decir que $L(M_w) \neq \Sigma^*$, que es lo que se necesita)

La figura siguiente ilustra la reducción planteada:



La función f es total computable.

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

Se cumple (<M>, w) $\in L_U \leftrightarrow$ f((<M>, w)) $\in L_{\Sigma^*}$.

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

Fin de Ejemplo

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

Ejemplo 4.5. Reducción de L_U^C a L_{Σ^*}

Se probará que L_{Σ^*} = {<M> | L(M) = Σ*} ∉ RE. Se hará L_U^C α L_{Σ^*}. Como $L_U \in$ RE – R, entonces $L_U^C \notin$ 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).

Definición de la función de reducción.

Para pares válidos (<M>, w) se define

f((<M>, w)) = <M_w>

donde M_w es una MT que a partir de una entrada v simula a lo sumo |v| pasos de M a partir de w (M podría detenerse antes), y acepta si y sólo si M no acepta w. Se comprueba fácilmente que:

- Si M no acepta w, entonces $L(M_w) = \Sigma^*$
- Si M acepta w, digamos en k pasos, entonces $L(M_w) = \{v \mid |v| < k\}$ (es decir que $L(M_w) \neq \Sigma^*$, que es lo que se necesita)

La función f es total computable.

Si la entrada no es una cadena válida (<M>, w), y establecemos por convención que la misma pertenece a L_U^C, entonces M_f genera un código $<M_{\Sigma^*}>$ tal que $L(M_{\Sigma^*}) = \Sigma^*$. En caso contrario, para generar $<M_w>$, la MT M_f le agrega al código <M> un fragmento que calcula el tamaño i de la entrada, decrementa i en 1 toda vez que se ejecuta un paso, detiene la ejecución cuando i = 0, y acepta si y sólo si no se alcanza al final el estado q_A.

Se cumple $(<M>, w) \in L_U^C \leftrightarrow f((<M>, w)) \in L_{\Sigma^*}$.

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

Fin de Ejemplo

De la reducción L_U α L_{Σ^*} se deduce, como vimos, la reducción L_U^C α $L_{\Sigma^*}^C$. Como además se cumple L_U^C α L_{Σ^*} y $L_U^C \notin$ RE, entonces tanto L_{Σ^*} como $L_{\Sigma^*}^C$ habitan la clase más difícil de la jerarquía de la computabilidad, es decir el conjunto Ω – (RE ∪ CO-RE).

Ejemplo 4.6. Reducción de L_{Σ^*} a L_{EQ}

Sea el problema: dadas dos MT M_1 y M_2, ¿M_1 es equivalente a M_2? Se quiere determinar si dicho problema es decidible. El lenguaje que lo representa es $L_{EQ} = \{(<M_1>, <M_2>) \mid L(M_1) = L(M_2)\}$. Hay que determinar si L_{EQ} es un lenguaje recursivo.

Intuitivamente, no parece ni siquiera que $L_{EQ} \in$ RE. Si lo fuera, existiría una MT M_{EQ} que acepta $(<M_1>, <M_2>)$ después de comprobar que M_1 y M_2 aceptan y rechazan exactamente las mismas cadenas 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. Se probará directamente que $L_{EQ} \notin$ RE. Se hará L_{Σ^*} α L_{EQ}. Como $L_{\Sigma^*} \notin$ RE, entonces $L_{EQ} \notin$ RE.

Definición de la función de reducción.

Para cadenas válidas <M>, se define

$f(<M>) = (<M>, <M_{\Sigma*}>)$

con $L(M_{\Sigma*}) = \Sigma*$.

La función f es total computable.

Si la entrada no es un código válido <M>, entonces la MT M_f genera la cadena 1. En caso contrario genera (<M>, $<M_{\Sigma*}>$), siendo $<M_{\Sigma*}>$ el código de una MT que acepta el lenguaje $\Sigma*$.

Se cumple $<M> \in L_{\Sigma*} \leftrightarrow f(<M>) \in L_{EQ}$.

$<M> \in L_{\Sigma*} \leftrightarrow L(M) = \Sigma* \leftrightarrow L(M) = L(M_{\Sigma*}) \leftrightarrow (<M>, <M_{\Sigma*}>) \in L_{EQ}$.

Fin de Ejemplo

Ejemplo 4.7. Reducciones de L_U^C a L_{REC} y L_{REC}^C

Probaremos que es indecidible determinar si el lenguaje aceptado por una MT es recursivo. El lenguaje que representa el problema es $L_{REC} = \{<M> \mid L(M) \in R\}$. Demostraremos directamente que L_{REC} no es recursivamente numerable. Se hará $L_U^C \alpha L_{REC}$. Como $L_U^C \notin RE$, entonces $L_{REC} \notin RE$.

Definición de la función de reducción.

Para pares válidos (<M>, w) se define

$f((<M>, w)) = <M_w>$

donde M_w es una MT que, a partir de una entrada v:

1. Simula M a partir de w.
2. Si M no acepta, entonces M_w no acepta.
3. Si M_U es una MT que acepta L_U, entonces M_w simula M_U a partir de v, y acepta si y sólo si M_U acepta.

Se comprueba fácilmente que:

- Si M no acepta w, entonces $L(M_w) = \varnothing$ (que es un lenguaje recursivo).
- Si M acepta w, entonces $L(M_w) = L_U$ (que no es un lenguaje recursivo).

La función f es total computable.

Si la entrada no es una cadena válida (<M>, w), la MT M_f genera un código $<M_\varnothing>$, con $L(<M_\varnothing>) = \varnothing$. En caso contrario genera $<M_w>$, básicamente agregándole a <M> un código $<M_U>$ y un fragmento que primero reemplaza la entrada v por la cadena w y luego la restituye para continuar la ejecución.

Se cumple $(<M>, w) \in L_U^C \leftrightarrow f((<M>, w)) \in L_{REC}$.

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

También se prueba que L_{REC}^C no es recursivamente numerable. Por lo tanto, al igual que L_{Σ^*}, el lenguaje L_{REC} habita la clase más difícil de la jerarquía de la computabilidad. Se hará L_U^C α L_{REC}^C. Como $L_U^C \notin RE$, entonces probaremos que $L_{REC}^C \notin RE$.

Definición de la función de reducción.

Para pares válidos (<M>, w) se define

$$f((<M>, w)) = <M_w>$$

donde M_w es una MT que, a partir de una entrada v, simula en paralelo M a partir de w y M_U a partir de v, aceptando si y sólo si alguna de las dos MT acepta. Se comprueba fácilmente que:

- Si M no acepta w, entonces $L(M_w) = L_U$ (que no es un lenguaje recursivo).
- Si M acepta w, entonces $L(M_w) = \Sigma^*$ (que es un lenguaje recursivo).

La función f es total computable.

Si la entrada no es una cadena válida (<M>, w), la MT M_f genera un código $<M_U>$, que pertenece a L_{REC}^C. En caso contrario, genera $<M_w>$ básicamente agregándole a <M>, un 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 $(\langle M \rangle, w) \in L_U^C \leftrightarrow f((\langle M \rangle, w)) \in L_{REC}^C$.

$(\langle M \rangle, w) \in L_U^C \leftrightarrow$ M no acepta w $\leftrightarrow L(M_w) = L_U \leftrightarrow \langle M_w \rangle \in L_{REC}^C$.

Fin de Ejemplo

Las reducciones de problemas presentadas se han basado en distintos modelos o "recetas", como por ejemplo:

- La generación de un código $\langle M_w \rangle$ a partir de una entrada $(\langle M \rangle, w)$, tal que la MT M_w ignora sus entradas y simula directamente M a partir de w.
- Una variante del modelo anterior, tal que la MT M_w simula primero M a partir de w, y después "filtra" sus entradas simulando otra MT M'. La elección de M' se relaciona con el lenguaje al que se pretende reducir.
- Otra variante del modelo anterior, tal que la MT M_w simula M a partir de w teniendo en cuenta alguna característica de sus entradas v, por ejemplo la longitud |v| para acotar los pasos simulados de M.

La elección de uno u otro modelo depende de los lenguajes que intervienen en las reducciones. En la segunda parte del libro se verán más casos.

El siguiente ejemplo de reducción de problemas se relaciona con un problema clásico de la lógica, ya referido, y parte de un problema sobre cadenas de símbolos, también mencionado previamente, que es el problema de correspondencia de Post (también conocido como PCP): dada una secuencia de pares de cadenas de unos y ceros no vacías $(s_1, t_1), \ldots, (s_k, t_k)$, ¿existe una secuencia de índices $i_1, \ldots, i_n$, con $n \geq 1$, tal que las cadenas $s_{i1} \ldots s_{in}$ y $t_{i1} \ldots t_{in}$ sean iguales? Por ejemplo, para los pares (1, 111), (10111, 10), (10, 0), se cumple que (2, 1, 1, 3) es solución: en ambos casos se obtiene la cadena 101111110. En cambio, para los pares (10, 101), (011, 11), (101, 011), se puede comprobar que no existe solución. Las soluciones pueden repetir índices; esto significa que el espacio de búsqueda con el que se trata es infinito, lo que es un indicio de la indecidibilidad del problema. Efectivamente, se prueba que existe una reducción de L_U a PCP, por lo que PCP es indecidible. No vamos a construir esta reducción, sino que utilizaremos PCP en otra reducción para probar, a continuación, que el problema de la validez en la lógica de primer orden (también conocido como VAL) es indecidible.

Ejemplo 4.8. Reducción de PCP a VAL

Sea VAL = {φ | φ es una fórmula válida de la lógica de primer orden}. Vamos a construir una reducción de PCP a VAL. Como PCP no es recursivo, entonces demostraremos que VAL tampoco lo es.

Dada una secuencia S de pares de cadenas de unos y ceros no vacías $(s_1, t_1), \ldots, (s_k, t_k)$, la función de reducción le asignará a S una fórmula φ de la lógica de primer orden válida si y sólo si S tiene solución, en el sentido del problema de correspondencia de Post.

Como símbolos de función utilizamos *e*, de aridad 0 (es una constante), y f_0 y f_1, de aridad 1. La idea es que e represente la cadena vacía, y f_0 y f_1 la concatenación, al final de una cadena, del dígito 0 o 1, respectivamente. De este modo, la cadena $b_1 \ldots b_m$ de dígitos binarios b_i se puede representar por el término $f_{bm}(\ldots(f_{b1}(e))\ldots)$, que para facilitar la notación lo abreviaremos con $f_{b1\ldots bm}(e)$.

Y como símbolo de predicado utilizamos P, de aridad 2; el significado entendido para P(s, t) es que existe una secuencia de índices $i_1, \ldots, i_n$, tal que el término s representa una cadena con subcadenas s_i de unos y ceros de la forma $s_{i1}\ldots s_{in}$, y el término t representa una cadena con subcadenas t_i de unos y ceros de la forma $t_{i1}\ldots t_{in}$.

Definición de la función de reducción.

La fórmula que se asigna, por la función de reducción f, a una secuencia S (sintácticamente correcta) de pares $(s_1, t_1), \ldots, (s_k, t_k)$, es

$\varphi = \varphi_1 \wedge \varphi_2 \rightarrow \varphi_3$, con:

$\varphi_1 = \bigwedge_{i=1,k} P(f_{si}(e), f_{ti}(e))$

$\varphi_2 = \forall v \forall w: P(v, w) \rightarrow \bigwedge_{i=1,k} P(f_{si}(v), f_{ti}(w))$

$\varphi_3 = \exists z: P(z, z)$

La función f es total computable.

Claramente, existe una MT M_f que dada una secuencia S sintácticamente correcta genera la fórmula φ descripta previamente (en otro caso M_f genera la cadena 1).

Se cumple $S \in PCP \leftrightarrow f(S) \in VAL$.

a. Supongamos primero que φ es válida, lo que se denota con $\models \varphi$. Vamos a probar que S tiene solución. Más específicamente, vamos a encontrar un modelo M_0 para φ que establezca la existencia de una secuencia de índices que soluciona S.

   El dominio de M_0 contiene todas las cadenas finitas de unos y ceros, incluyendo la cadena vacía λ. La interpretación de *e* es λ, lo que se denota con $e^{M0} = \lambda$. La interpretación de f_0 es la concatenación de un 0 al final de una cadena, lo que se denota con $f_0^{M0}(s) = s0$. De la misma manera se define $f_1^{M0}(s) = s1$. Finalmente, la interpretación de P es la siguiente: se cumple $P^{M0}(s, t)$ cuando existe una secuencia de índices $i_1, \ldots, i_n$, tal que $s = s_{i1}\ldots s_{in}$, $t = t_{i1}\ldots t_{in}$, y s_i y t_i son cadenas de unos y ceros de S.

   Como vale $\models \varphi$, se cumple en particular $M_0 \models \varphi$. Claramente vale $M_0 \models \varphi_1$, porque se cumple $P^{M0}(s_i, t_i)$ para $i = 1, \ldots, k$. Veamos que también vale $M_0 \models \varphi_2$, que establece que cuando el par (s, t) está en P^{M0}, también lo está el par (ss_i, tt_i), para $i = 1, \ldots, k$. Si $(s, t) \in P^{M0}$, entonces existe una secuencia de índices $i_1, \ldots, i_n$, tal que $s = s_{i1}\ldots s_{in}$ y $t = t_{i1}\ldots t_{in}$. Definiendo una nueva secuencia de índices $i_1, \ldots, i_n$, i, vale $ss_i = s_{i1}\ldots s_{in}s_i$ y $tt_i = t_{i1}\ldots t_{in}t_i$, por lo que se cumple $M_0 \models \varphi_2$. De esta manera, como se cumple $M_0 \models \varphi_1 \wedge \varphi_2 \rightarrow \varphi_3$ por hipótesis y hemos demostrado recién $M_0 \models \varphi_1 \wedge \varphi_2$, vale $M_0 \models \varphi_3$. Finalmente, por la definición de φ_3 y P^{M0}, existe una solución para S.

b. Supongamos ahora que S tiene solución, digamos la secuencia de índices $i_1, \ldots, i_n$. Vamos a probar que cualquiera sea el modelo M, con una constante e^M, dos funciones unarias f_0^M y f_1^M, y un predicado binario P^M, entonces M satisface φ, es decir $M \models \varphi$. Dada la fórmula $\varphi = \varphi_1 \wedge \varphi_2 \rightarrow \varphi_3$, vamos a asumir $M \models \varphi_1 \wedge \varphi_2$ y demostraremos $M \models \varphi_3$.

   Para la interpretación de las cadenas finitas de unos y ceros en el dominio de M definimos inductivamente una función denominada *interpret*, de la siguiente manera: $interpret(\lambda) = e^M$, $interpret(s0) = f_0^M(interpret(s))$, e $interpret(s1) = f_1^M(interpret(s))$. Por ejemplo, a la cadena 110 se le asigna $f_0^M(f_1^M(f_1^M(e^M)))$.

   Más genéricamente, si una cadena s de dígitos binarios b_i tiene la forma $b_{1\ldots}b_m$, entonces $interpret(b_1\ldots b_m) = f_{bm}^M(\ldots(f_{b1}^M(e^M))\ldots)$, abreviado con $f_s^M(e)$ como indicamos antes. Entonces, como vale $M \models \varphi_1$, se cumple $(interpret(s_i), interpret(t_i)) \in P^M$, para $i = 1, \ldots, k$. Como también vale $M \models \varphi_2$, entonces para

todo par (s, t) ∈ P^M se cumple (interpret(ss_i), interpret(tt_i)) ∈ P^M, para i = 1, …, k. De esta manera, comenzando con (s, t) = (s_{i1}, t_i), si se considera repetidamente la última observación se obtiene (interpret($s_{i1}…s_{in}$), interpret($t_{i1}…t_{in}$)) ∈ P^M, y dado que las cadenas $s_{i1}…s_{in}$ y $t_{i1}…t_{in}$ son iguales porque i_1, …, i_n es una solución de S, entonces interpret($s_{i1}…s_{in}$) = interpret($t_{i1}…t_{in}$). De este modo se cumple la fórmula ∃z: P(z, z), y así M |= φ_3.

Fin de ejemplo

Como una fórmula es insatisfactible si y sólo si su negación es válida, del ejemplo anterior se desprende que el lenguaje de las fórmulas satisfactibles de la lógica de primer orden también es indecidible (la prueba queda como ejercicio). Lo mismo sucede con el lenguaje de los teoremas, porque por la sensatez y completitud de la lógica de primer orden dicho lenguaje coincide con el de las fórmulas válidas. Por otra parte, el lenguaje de los teoremas (y de las fórmulas válidas) es recursivamente numerable: a partir de axiomas y reglas de inferencia se puede construir fácilmente una MT que lo reconoce (la prueba queda como ejercicio).

La indecidibilidad en la lógica de primer orden contrasta con la decidibilidad en la lógica proposicional, en la que la satisfactibilidad y validez de las fórmulas se determinan mediante las tablas de verdad.

Por su parte, mientras el lenguaje de los teoremas de la teoría de números es recursivamente numerable, el de las fórmulas verdaderas no lo es: por el Teorema de Incompletitud de Gödel, toda teoría recursiva y consistente que contenga "suficiente" aritmética es incompleta. En particular, la indecidibilidad en la teoría de números se puede probar mediante una reducción a partir del problema de pertenencia de la cadena vacía λ en un lenguaje recursivamente numerable, que se demuestra es indecidible. La idea de la reducción es la siguiente. El lenguaje de la teoría de números, si cuenta con las operaciones de suma y multiplicación ("suficiente" aritmética o expresividad), como es el caso de la aritmética de Peano, permite expresar las configuraciones y computaciones de las MT mediante números. Si una MT M acepta λ, lo hace con una computación en la que ninguna configuración mide, naturalmente, más que un número determinado k. La reducción asigna entonces a cada código <M> una fórmula ∃i∃k(φ_k(i)), que es verdadera si y sólo si i representa una computación de M que acepta λ con configuraciones que no miden más que k.

A diferencia de la teoría de números, la aritmética sin la multiplicación, conocida como aritmética de Presburger, es decidible. Otros ejemplos de teorías decidibles son las teorías de los números reales y los números complejos.

Como último tema de esta clase presentamos un mecanismo muy sencillo para probar la no recursividad de un determinado tipo de lenguajes. Se trata del Teorema de Rice, que se demuestra en lo que sigue por medio de una reducción de problemas. El teorema establece que todo lenguaje que representa una propiedad de los lenguajes recursivamente numerables no es recursivo (salvo que la propiedad sea trivial e involucre a todos los lenguajes de RE o a ninguno). Ejemplos de propiedades no triviales son ser finito, infinito, recursivo, no recursivo, tener un número par de cadenas, ser el lenguaje $\varnothing$, el lenguaje Σ^*, etc. De esta manera, podremos determinar inmediatamente por la aplicación del Teorema de Rice que por ejemplo el lenguaje L = {<M> | L(M) es recursivo}, que ya tratamos previamente, no es recursivo.

Teorema 4.2. Teorema de Rice

Vamos a demostrar que si $\mathcal{P}$ es un subconjunto de RE, con $\varnothing \subset \mathcal{P} \subset$ RE, entonces el lenguaje $L_{\mathcal{P}}$ = {<M> | L(M) $\in \mathcal{P}$} no es recursivo. $\mathcal{P}$ representa una propiedad no trivial de RE. Intuitivamente, es razonable que $L_{\mathcal{P}}$ no sea recursivo: probar que los lenguajes L(M) cumplen una determinada propiedad requiere chequear, en cada caso, el comportamiento de M sobre las infinitas cadenas de Σ^*. El teorema no dice nada con respecto a si $L_{\mathcal{P}}$ está o no en RE (el comentario anterior ahora no aplica, porque en el caso de los lenguajes recursivamente numerables las MT que los reconocen pueden no detenerse). Existe de todos modos una variante del teorema que también considera el caso RE, pero no la describiremos. El teorema tampoco dice 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 determinado número de estados, que no recorren más de un cierto número de celdas, que sólo se mueven a la derecha, que escriben alguna vez un símbolo dado, etc).

Supongamos primero que $\varnothing \notin \mathcal{P}$. Sea $L_1 \neq \varnothing$ un lenguaje de $\mathcal{P}$, y M_1 una MT que reconoce L_1. Se va a construir una reducción de L_U α $L_{\mathcal{P}}$, y de esta manera se probará que $L_{\mathcal{P}} \notin$ R.

Definición de la función de reducción.

Para pares válidos (<M>, w) se define

$$f((<M>, w)) = <M_w>$$

donde M_w es una MT que a partir de una entrada v:

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 si y sólo si M_1 acepta.

Se comprueba fácilmente que:

- Si M acepta w, entonces $L(M_w) = L_1$ (que pertenece a $\mathcal{P}$).
- Si M no acepta w, entonces $L(M_w) = \varnothing$ (que no pertenece a $\mathcal{P}$).

La función f es total computable.

Si la entrada no es una cadena válida (<M>, w), la MT M_f genera la cadena 1 (por convención en este caso $L(M_w) = \varnothing$). En caso contrario, M_f genera $<M_w>$ básicamente 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 $(<M>, w) \in L_U \leftrightarrow f((<M>, w)) \in L_{\mathcal{P}}$.

$(<M>, w) \in L_U \leftrightarrow$ M acepta w $\leftrightarrow L(M_w) = L_1 \in \mathcal{P} \leftrightarrow <M_w> \in L_{\mathcal{P}}$.

Supongamos ahora que $\varnothing \in \mathcal{P}$. Sea $\mathcal{P}' = RE - \mathcal{P}$. Como $\varnothing \notin \mathcal{P}'$, con $\varnothing \subset \mathcal{P}' \subset RE$, entonces por lo probado recién, se cumple $L_{\mathcal{P}'} \notin R$. Pero $L_{\mathcal{P}'} = L_{\mathcal{P}}{}^C$. Por lo tanto, en este caso también vale $L_{\mathcal{P}} \notin R$.

Fin de Teorema

Los siguientes ejemplos tienen el objetivo principal de clarificar cuándo y cómo se puede aplicar el Teorema de Rice para probar la no recursividad de un lenguaje.

Ejemplo 4.9. $L_{par} \notin R$ (aplicación del Teorema de Rice)

Sea L_{par} = {<M> | |L(M)| es par}. Se prueba que $L_{par} \notin R$ por aplicación del Teorema de Rice:

1. La propiedad de RE representada por el lenguaje L_{par} es $\mathcal{P}$ = {L ∈ RE | |L| es par}, es decir, es la propiedad de tener un número par de cadenas.
2. Se cumple $\varnothing \subset \mathcal{P}$ porque por ejemplo {a, b} ∈ $\mathcal{P}$.
3. Se cumple $\mathcal{P} \subset$ RE porque por ejemplo {c} ∉ $\mathcal{P}$.

De esta manera se prueba que $L_{par} \notin R$.

Fin de Ejemplo

Ejemplo 4.10. $L_{\varnothing} \notin R$ (aplicación del Teorema de Rice)

Sea $L_{\varnothing}$ = {<M> | L(M) = ∅}. Se prueba que $L_{\varnothing} \notin R$ por aplicación del Teorema de Rice:

1. La propiedad de RE representada por el lenguaje $L_{\varnothing}$ es $\mathcal{P}$ = {∅}, es decir, es la propiedad de ser el lenguaje vacío.
2. Se cumple $\varnothing \subset \mathcal{P}$ porque ∅ ∈ $\mathcal{P}$.
3. Se cumple $\mathcal{P} \subset$ RE porque por ejemplo Σ* ∉ $\mathcal{P}$.

De esta manera se prueba que $L_{\varnothing} \notin R$.

Fin de Ejemplo

El problema de determinar si una MT reconoce el lenguaje vacío es otro problema clásico de la computabilidad. Para completar su ubicación en la jerarquía vamos a probar que $L_{\varnothing}$ no es ni siquiera recursivamente numerable. No lo haremos por medio de una reducción de problemas (queda como ejercicio), sino que demostraremos que $L_{\varnothing}^{C} \in$ RE, y así probaremos que $L_{\varnothing} \notin$ RE (de lo contrario $L_{\varnothing}$ sería recursivo). Dado el lenguaje $L_{\varnothing}^{C}$ = {<M> | L(M) ≠ ∅}, construimos a continuación una MT $M_{\varnothing}^{C}$ que lo reconoce. A partir de una entrada w, la MT $M_{\varnothing}^{C}$ trabaja de la siguiente manera:

1. Si w = <M> es inválido, rechaza.
2. Hace i := 0.
3. Simula M a lo sumo i pasos a partir de todas las cadenas v tales que $|v| \leq i$. Si en algún caso M acepta, acepta.
4. Hace i := i + 1 y vuelve al paso 3.

Se cumple que $L_{\varnothing}^{C} = L(M_{\varnothing}^{C})$: <M> $\in L_{\varnothing}^{C} \leftrightarrow$ L(M) tiene al menos una cadena v $\leftrightarrow$ $M_{\varnothing}^{C}$ detecta en algún paso que M acepta alguna cadena v $\leftrightarrow$ $M_{\varnothing}^{C}$ acepta <M> $\leftrightarrow$ <M> $\in L(M_{\varnothing}^{C})$.

Los últimos ejemplos corresponden a casos en que el Teorema de Rice no es aplicable. Para determinar si los lenguajes son o no son recursivos se debe recurrir a otros caminos.

Ejemplo 4.11. L_{20} es recursivo

Sea L_{20} = {<M> | M es una MT determinística con una cinta, y a partir de la entrada vacía λ nunca sale de las celdas 1 a 20}. Por convención, se define que la celdas se numeran 1, 2, 3, etc., de izquierda a derecha, a partir de la celda apuntada inicialmente por el cabezal. En este caso no se puede aplicar el Teorema de Rice, porque los códigos <M> no se definen en términos de L(M). Vamos a probar que $L_{20} \in R$.

Si <M> $\in L_{20}$, entonces existe un número máximo C de configuraciones distintas por las que pasa M, con |Q| estados y |Γ| símbolos, antes de entrar en un *loop*. Más precisamente, $C = 20.|Q|.|\Gamma|^{20}$. 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 un *loop*. La MT M_{20} tiene cinco cintas. A partir de una entrada w trabaja de la siguiente manera:

1. Si w no es un código <M> válido, rechaza.
2. Calcula y escribe C en la cinta 5.
3. Hace i := 1 en la cinta 3. La variable i identifica la posición del cabezal de M.
4. Hace n := 0 en la cinta 4. La variable n identifica el número de pasos ejecutados por M.
5. Simula el paso siguiente de M con entrada λ, en la cinta 2.

6. Actualiza adecuadamente el valor de i en la cinta 3. Si i = 0 o i = 21, se detiene en q_R. Si M se detuvo, se detiene en q_A.
7. Hace n := n + 1 en la cinta 4. Si n = C, se detiene en q_A.
8. Vuelve al paso 5.

En el paso 2 la cota C se calcula a partir de los valores |Q| y |Γ| que se obtienen de <M>. En el paso 6 el valor de i se incrementa en 1, se decrementa en 1 o se mantiene, si M se mueve a derecha, izquierda o no se mueve, respectivamente. M_{20} se detiene siempre porque alguna de las tres posibilidades de detención indicadas en el algoritmo debe cumplirse, cualquiera sea <M>. Se cumple además que $L_{20} = L(M_{20})$: $<M> \in L_{20} \leftrightarrow$ M a partir de λ no sale de las celdas 1 a 20 $\leftrightarrow M_{20}$ se detiene a partir de <M> en su estado $q_A \leftrightarrow <M> \in L(M_{20})$.

Fin de Ejemplo

De esta manera, limitando el espacio de trabajo de una MT se puede acotar también su número de pasos, es decir su tiempo de ejecución, teniendo en cuenta la cantidad de configuraciones distintas que puede recorrer. La relación entre el espacio y el tiempo consumidos por una MT se trata con cierto detalle en la Clase 9.

Ejemplo 4.12. L_{imp0} no es recursivo

Sea L_{imp0} = {<M> | M es una MT que a partir de toda entrada escribe alguna vez el símbolo 0}. Como los códigos <M> no se definen en términos de L(M), tampoco en este caso puede aplicarse el Teorema de Rice. Probaremos que $L_{imp0} \notin R$, mediante una reducción de problemas de HP a L_{imp0}.

Definición de la función de reducción.
Para pares válidos (<M>, w) se define

$$f((<M>, w)) = <M_w>$$

tal que $<M_w>$ tiene las siguientes características:

- Un primer fragmento borra la entrada y la reemplaza por la cadena w', tal que w' es como w salvo que en lugar del símbolo 0 tiene un símbolo x que no existe en el alfabeto de M.
- Un segundo fragmento tiene las mismas tuplas que M, salvo que en lugar del símbolo 0 utiliza el símbolo x, y en lugar de los estados finales q_A y q_R utiliza, respectivamente, nuevos estados no finales q_A' y q_R'.
- Un último fragmento tiene nuevas tuplas definidas a partir de los estados q_A' y q_R' y todos los símbolos z del alfabeto de M_w menos el símbolo 0, de la forma $\delta(q_A', z) = (q_A, 0, S)$ y $\delta(q_R', z) = (q_A, 0, S)$.

La idea es que M_w replique los pasos de M sin escribir nunca el símbolo 0, y sólo en el caso de que M se detenga, M_w haga un paso más escribiendo el 0.

La función f es total computable.

La función de reducción es claramente total computable (si la entrada no es una cadena válida (<M>, w), la MT M_f genera la cadena 1).

Se cumple (<M> , w) $\in$ HP $\leftrightarrow$ <M_w> $\in$ L_{imp0}.

(<M>, w) $\in$ HP $\leftrightarrow$ M se detiene a partir de w $\leftrightarrow$ M_w a partir de toda entrada escribe el símbolo 0 $\leftrightarrow$ <M_w> $\in$ L_{imp0}.

Fin de Ejemplo

Ejercicios de la Clase 4

1. Probar que las reducciones de problemas son reflexivas y transitivas.
2. Probar que si $L_2 \in$ RE y L_1 α L_2, entonces $L_1 \in$ RE.
3. Probar que todo lenguaje de la clase RE se puede reducir a HP.
4. Construir una reducción:
   i. De HP al lenguaje de las ternas (<M>, w, v), tales que M(w) = v, es decir que la MT M a partir de la cadena w genera la cadena v.
   ii. De L_U al lenguaje de los pares (<M>, w), de modo tal que para todo par (<M>, w) existe una cadena v que cumple M(w) = v.
5. Determinar a qué conjuntos (R, RE – R, CO-RE – R, Ω – (RE ∪ CO-RE)), pertenecen los siguientes lenguajes:

i. $L = \{<M> \mid M \text{ se detiene a partir de todas las cadenas}\}$.

ii. $L = \{<M> \mid M \text{ se detiene a partir de alguna cadena}\}$.

iii. $L = \{<M> \mid M \text{ no se detiene a partir de alguna cadena}\}$.

iv. $L = \{<M> \mid M \text{ no se detiene a partir de ninguna cadena}\}$.

6. Sea w una cadena de unos y ceros, y E(w) la cadena generada reemplazando en w los ceros por unos y los unos por ceros. Por ejemplo, E(1001) = 0110. Se define que L es un lenguaje espejo, si toda cadena $w \in L$ distinta de la cadena vacía λ cumple que $E(w) \in L^C$. Probar que si L es un lenguaje espejo que no pertenece a RE, entonces L^C tampoco pertenece a RE.
7. Probar que no existe una reducción de $L_{\Sigma^*} = \{<M> \mid L(M) = \Sigma^*\}$ a $L_\varnothing = \{<M> \mid L(M) = \emptyset\}$.
8. Determinar si los siguientes lenguajes son recursivos, recursivamente numerables o no recursivamente numerables:

   i. $L = \{<M> \mid \exists w \text{ tal que M se detiene a partir de w en a lo sumo 1000 pasos}\}$.

   ii. $L = \{<M> \mid \exists w \in L(M) \text{ tal que } |w| < 100\}$.

   iii. $L = \{<M> \mid M \text{ es una MT con un número impar de estados}\}$.

   iv. $L = \{<M> \mid L(M) = S\}$, con $S \subseteq \Sigma^*$.

   v. $L = \{<M> \mid L(M) \text{ es finito}\}$.

   vi. $L = \{<M> \mid L(M) \in RE\}$.

   vii. $L = \{<M> \mid L(M) \in RE - R\}$.

   viii. $L = \{<M> \mid \lambda \in L(M)\}$.

   ix. $L = \{<M> \mid w \in L(M) \text{ si y sólo si } w^R \in L(M)\}$.

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

   xi. $L = \{(<M_1>, <M_2>) \mid L(M_1) \subseteq L(M_2)\}$.
9. Indicar si los siguientes problemas son decidibles:

   i. Dado el par (<M>, w), determinar si M a partir de w recorre todos sus estados.

   ii. Dado el código <M>, determinar si M tiene un estado inalcanzable, es decir un estado al que nunca llegan sus computaciones.

   iii. Dado el código <M>, determinar si M siempre escribe un símbolo blanco sobre un símbolo no blanco.
10. Probar que la lógica proposicional es decidible.
11. Los siguientes axiomas y reglas de inferencia sistematizan la lógica de primer orden y la teoría de números o aritmética:

Axiomas y reglas de la lógica de primer orden (con igualdad)

A1: $P \rightarrow (Q \rightarrow P)$

A2: $(P \rightarrow (Q \rightarrow R)) \rightarrow ((P \rightarrow Q) \rightarrow (P \rightarrow R))$

A3: $(\neg P \rightarrow \neg Q) \rightarrow (Q \rightarrow P)$

A4: $\forall x P(x) \rightarrow P[x \mid t]$, tal que la variable x y las variables del término t aparecen libres en P

A5: $\forall x(P \rightarrow Q) \rightarrow (P \rightarrow \forall x Q)$, tal que la variable x no aparece libre en P

A6: $x = x$

A7: $x = y \rightarrow y = x$

A8: $x = y \rightarrow (y = z \rightarrow x = z)$

A9: $x = y \rightarrow t(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_n) = t(v_1, \ldots, v_{i-1}, y, v_{i+1}, \ldots, v_n)$

A10: $x = y \rightarrow F(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_n) = F(v_1, \ldots, v_{i-1}, y, v_{i+1}, \ldots, v_n)$, tal que F es una fórmula atómica

R1: Modus Ponens. Si P y $(P \rightarrow Q)$ entonces Q

R2: Generalización. Si P entonces $\forall x P$

Axiomas de la aritmética (de Peano)

A11: $\neg(0 = S(x))$, donde S es la función sucesor

A12: $S(x) = S(y) \rightarrow x = y$

A13: $x + 0 = x$

A14: $x + S(y) = S(x + y)$

A15: $x \cdot 0 = 0$

A16: $x \cdot S(y) = (x \cdot y) + x$

A17: Inducción. Dada la fórmula o propiedad F, si x aparece libre en $F(x, v_1, \ldots, v_n)$, entonces $F(0, v_1, \ldots, v_n) \rightarrow (\forall x(F(x, v_1, \ldots, v_n) \rightarrow F(S(x), v_1, \ldots, v_n)) \rightarrow \forall x F(x, v_1, \ldots, v_n))$

En este marco, probar los siguientes incisos, considerando lo tratado en la Clase 4:

i. El lenguaje de los teoremas de la lógica de primer orden es recursivamente numerable y no es recursivo.

ii. El lenguaje de las fórmulas satisfactibles de la lógica de primer orden no es recursivo.

iii. Los lenguajes de las fórmulas verdaderas y de las fórmulas falsas de la aritmética no son recursivamente numerables.

12. Se demuestra que las fórmulas insatisfactibles de la lógica de primer orden y los teoremas de la aritmética son recursivamente inseparables (ver definición en el Ejercicio 9 de la Clase 3). Teniendo en cuenta este enunciado probar nuevamente los incisos del ejercicio anterior.

Clase 5. Misceláneas de computabilidad

TEMA 5.1. MÁQUINAS RAM. EQUIVALENCIA CON LAS MÁQUINAS DE TURING.

Las *máquinas RAM* (por *random access machines* o máquinas de acceso aleatorio) constituyen otro conocido modelo de computación, más cercano a las computadoras reales, y por eso resulta interesante probar que son equivalentes a las MT, es decir que tienen el mismo poder computacional. 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 R_1, R_2, R_3, …, que también pueden almacenar enteros. En las palabras de memoria hay instrucciones y datos. Las instrucciones son LOAD, STORE, ADD, etc., con la semántica conocida.
Vamos a probar primero que dada una MT M con una cinta semi-infinita, existe una RAM R equivalente (ya se demostró en el Ejemplo 1.4 la equivalencia entre los modelos de las MT con cintas infinitas y cintas semi-infinitas). Sólo presentamos la idea general. Las características de la RAM R son:

- El registro R_1 contiene en todo momento la dirección de la palabra de memoria que almacena el símbolo corriente de la MT M, representado en binario.
- El registro R_2 contiene en todo momento el estado corriente de la MT M, representado en binario.
- Sea $[R_1]$ el contenido de la palabra de memoria apuntada por la dirección almacenada en R_1. Para simplificar la escritura, en lugar de las instrucciones LOAD, STORE, ADD, etc., utilizaremos instrucciones de mayor nivel de abstracción, fácilmente implementadas por las mismas. Si por ejemplo en la MT M se tiene $\delta(q, x) = (q', x', L)$, entonces en la RAM R hay una instrucción de la forma siguiente:

  if (R_2 = q and $[R_1]$ = x) then R_2 := q'; $[R_1]$:= x'; R_1 := R_1 – 1 fi
- La instrucción principal de la RAM R es una iteración, cuyo cuerpo es una selección múltiple que considera todos los pares (q, x) de la MT M. La iteración termina si y sólo si la MT M se detiene.

Ahora desarrollamos la idea general de la prueba en el sentido contrario. Dada una RAM R, veamos que existe una MT M equivalente con varias cintas semi-infinitas. En la cinta 1 están primero las palabras de memoria de R con contenido (las instrucciones y la entrada), y luego hay símbolos blancos. Por ejemplo, usando notación binaria, el contenido de la cinta 1 empezaría así:

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

donde v_i es el contenido, en binario, de la palabra de memoria i, es decir la palabra de memoria apuntada por la dirección i. Los distintos registros aritméticos de la RAM R están en las cintas 2, 3, 4, …, de la MT M. La MT M tiene dos cintas más:

- Una cinta para almacenar el *location counter* (o contador de locación, abreviado con LC) de la RAM R, que contiene en todo momento la dirección de la palabra de memoria de la que debe leerse la próxima instrucción. Al comienzo vale cero.
- Una cinta para almacenar el *memory address register* (o registro de dirección de memoria, abreviado con MAR) de la RAM R, que contiene en todo momento la dirección de la palabra de memoria de la que debe leerse el próximo dato.

Supongamos que los primeros dígitos binarios del código de una instrucción representan el tipo de instrucción (puede ser LOAD, STORE, ADD, etc.), y que los dígitos restantes representan la dirección del operando referenciado por la instrucción. En lo que sigue, se presenta un ejemplo concreto para describir la simulación de la RAM R por medio de la MT M. Si en un momento dado la cinta de la MT M correspondiente al LC tiene el valor i, M trabaja de la siguiente manera:

1. Recorre su cinta 1 desde el extremo izquierdo y busca la cadena #i*.
2. Si encuentra antes un símbolo blanco, significa que no hay ninguna instrucción en la palabra de memoria i, es decir que la RAM R terminó, por lo que se detiene.
3. Procesa los dígitos siguientes a #i* hasta el próximo símbolo #, es decir que procesa v_i. Supongamos, por ejemplo, que los primeros dígitos de v_i representan

la instrucción ADD TO R_2, y que los dígitos restantes representan el número k. Entonces, la MT M:

3.1 Suma 1 al valor de la cinta del LC (actualización para la próxima instrucción).
3.2 Copia k en la cinta del MAR (dirección de la palabra de memoria con el valor a sumar al registro R_2).
3.3 Busca la cadena #k* en la cinta 1 desde el extremo izquierdo. Si #k* no aparece, no hace nada (se asume que el valor buscado es cero). En caso contrario obtiene v_k de la cadena #k*v_k#, y lo suma al contenido de la cinta que representa el registro R_2.

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

TEMA 5.2. MÁQUINAS DE TURING GENERADORAS DE LENGUAJES. GRAMÁTICAS.

Las MT que generan lenguajes tienen una *cinta de salida de sólo escritura,* en la que el cabezal se mueve únicamente hacia la derecha, y las cadenas se separan por el símbolo #. Se asume que la entrada es la cadena vacía λ.

El lenguaje generado por una MT es el conjunto de cadenas que escribe en su cinta de salida. Las cadenas se pueden repetir, y no aparecen en ningún orden determinado.

La expresión G(M) denota el lenguaje generado por la MT M. De esta manera, utilizando las expresiones G(M) y L(M) puede identificarse qué visión de MT se está considerando, si generadora o reconocedora, respectivamente.

Vamos a probar a continuación que las dos visiones son equivalentes, y por lo tanto, que otra condición suficiente y necesaria para que un lenguaje sea recursivamente numerable es que exista una MT que lo genere, lo que explica la denominación de esta clase de lenguajes.

Teorema 5.1. Equivalencia entre las MT reconocedoras y generadoras

Se cumple que existe una MT M_1 que reconoce un lenguaje L si y sólo si existe una MT M_2 que lo genera, es decir: $L \in RE \leftrightarrow \exists M: L = G(M)$.

Probamos primero que si $L \in RE$, entonces $\exists M: L = G(M)$. Sea M_1 una MT tal que $L(M_1) = L$. La idea es construir una MT M_2 con las siguientes características. M_2 genera, una a una, todas las cadenas w de Σ^* en el orden canónico, y a partir de cada w simula M_1. Cuando M_1 acepta w, entonces M_2 la escribe en su cinta de salida. Hay que tener en cuenta que M_1 puede entrar en un *loop* a partir de determinadas w, por lo que las simulaciones llevadas a cabo por M_2 deben hacerse de una manera particular. Concretamente, M_2 trabaja de la siguiente manera:

1. Hace i := 0.
2. Genera todas las cadenas w de Σ^* de longitud a lo sumo i en el orden canónico.
3. Simula a lo sumo i pasos de M_1 a partir de cada w generada en el paso 2.
4. Escribe en su cinta de salida las w aceptadas en el paso 3.
5. Hace i := i + 1.
6. Vuelve al paso 2.

M_2 no se detiene aún si L es finito. Una misma cadena se puede escribir más de una vez en su cinta de salida (se puede modificar la construcción evitando las repeticiones, por medio de una cinta auxiliar en la que se almacenan las cadenas que se van escribiendo). El orden de escritura de las cadenas depende de las características de M_1. Queda como ejercicio probar que $G(M_2) = L$.

Demostramos ahora el sentido contrario: si $\exists M: L = G(M)$, entonces $L \in RE$. Sea M_1 una MT tal que $G(M_1) = L$. La idea es construir una MT M_2 con las siguientes características. M_2 tiene una cinta más que M_1, en la que está su entrada v. Trabaja como M_1, pero toda vez que M_1 escribe una cadena w en su cinta de salida, M_2 la compara con v, y si son iguales acepta. Por otro lado, si M_1 se detiene en algún momento, entonces M_2 rechaza. Queda como ejercicio probar que $L(M_2) = L$.

En particular, los lenguajes recursivos se pueden generar en el orden canónico y sin repeticiones. Veamos primero que si L es un lenguaje recursivo, existe una MT M que lo genera en el orden canónico y sin repeticiones. Sea M_1 una MT tal que $L(M_1) = L$ y M_1 se detiene siempre. La idea es construir una MT M_2 de la siguiente manera. Como

antes, M_2 genera una a una todas las cadenas w de Σ^* en el orden canónico, y a partir de cada w simula M_1. Cuando M_1 acepta w, entonces M_2 la escribe en su cinta de salida. Ahora las simulaciones se pueden efectuar completamente, una después de la otra, porque M_1 se detiene siempre. Queda como ejercicio probar que M_2 genera L en el orden canónico y sin repeticiones.

Demostramos ahora el sentido contrario, es decir que si existe una MT que genera un lenguaje L en el orden canónico y sin repeticiones, entonces L es recursivo. Sea M_1 una MT con dichas características. La idea es construir una MT M_2 del siguiente modo. Como antes, M_2 tiene una cinta más que M_1, en la que está su entrada v, y trabaja como M_1, salvo que toda vez que M_1 escribe una cadena w en su cinta de salida, M_2 hace:

1. Si w = v, acepta.
2. Si w > v según el orden canónico, rechaza (porque ya no hay posibilidad de que M_1 imprima v en su cinta de salida).
3. Si w < v según el orden canónico, espera por la próxima escritura de M_1 en su cinta de salida (porque es posible que la cadena v aparezca después).

Por otro lado, si M_1 se detiene en algún momento, entonces M_2 rechaza. Notar que esta última construcción sirve sólo cuando L es infinito. En efecto, si L es finito y la MT M_1 que lo genera no se detiene, puede suceder que M_2 tampoco se detenga: es el caso en que la entrada v de M_2 es mayor que la mayor cadena de L según el orden canónico, digamos u, porque M_2 esperará infructuosamente después que M_1 haya escrito la cadena u. Pero si L es finito se prueba que es recursivo (ver Ejercicio 3 de la Clase 2), sin necesidad de recurrir a la construcción anterior. Queda como ejercicio probar que $L(M_2) = L$ y que M_2 se detiene siempre cuando L es infinito.

Fin de Teorema

Otra conocida representación de los lenguajes con la visión generadora la constituyen las *gramáticas*.

Definición 5.1. Gramática

Una gramática es una 4-tupla $G = (V_N, V_T, P, S)$, tal que:

- V_N es un conjunto de símbolos denominados *no terminales*.
- V_T es un conjunto de símbolos denominados *terminales*, disjuntos de los de V_N. El conjunto $V_N \cup V_T$ se denota con V.
- P es un conjunto de reglas denominadas *producciones*. Una producción tiene la forma $\alpha \rightarrow \beta$, con $\alpha \in V^+$ y $\beta \in V^*$.
- Finalmente, S es el *símbolo inicial* de G (también se lo conoce como *axioma*), siendo $S \in V_N$.

Fin de Definición

La generación de un lenguaje a partir de una gramática G se establece en base a una relación identificada con $\Rightarrow_G$. Si por ejemplo $\alpha \rightarrow \beta$ es una producción de G, y las cadenas γ y δ están en V*, entonces se cumple que $\gamma\alpha\delta \Rightarrow_G \gamma\beta\delta$. La expresión $\alpha_1 \Rightarrow^*_G \alpha_m$ indica que se cumple $\alpha_1 \Rightarrow_G \alpha_2$, …, $\alpha_{m-1} \Rightarrow_G \alpha_m$. Se dice en este caso que existe una derivación de α_1 a α_m, o que α_m deriva de α_1. El lenguaje generado por una gramática G se denota con la expresión L(G), y se define así: $L(G) = \{w \mid w \in V_T^* \text{ y } S \Rightarrow^*_G w\}$. Es decir que L(G) es el conjunto de todas las cadenas de V_T^* que derivan del símbolo inicial S de la gramática G.

Ejemplo 5.1. Gramática del lenguaje de las cadenas 0^n1^n, con $n \geq 1$

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

Fin de Ejemplo

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

1. *Gramática de tipo 0* o *semi-Thue*. No tiene ninguna restricción.
2. *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 $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, con $A \in V_N$, α_1 y $\alpha_2 \in V^*$, y $\beta \in V^+$.

3. *Gramática de tipo 2 o libre de contexto*. Toda producción es de la forma $A \rightarrow \alpha$, con $A \in V_N$, y $\alpha \in V^+$, lo que explica el nombre del tipo de gramática.
4. *Gramática de tipo 3 o regular*. Toda producción es de la forma $A \rightarrow aB$ o bien $A \rightarrow a$, con 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 la cadena vacía 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 si y sólo si es recursivamente numerable: a partir de una gramática G semi-Thue se puede construir una MT que reconoce L(G) y viceversa. También se prueba que los lenguajes sensibles al contexto son recursivos, y 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 sensible al contexto no tiene menos símbolos que la parte izquierda, por lo que siempre se puede determinar si existe una derivación del símbolo inicial a una cadena de símbolos terminales. Formalmente, dada una gramática sensible al contexto $G = (V_N, V_T, P, S)$ y una cadena w de longitud n, se puede determinar si $w \in L(G)$ de la siguiente manera. Se construye un grafo orientado cuyos vértices son todas las cadenas de V* de longitud a lo sumo n, y cuyos arcos son todos los pares (α, β) tales que se cumple $\alpha \Rightarrow_G \beta$. De esta manera, los caminos del grafo se corresponden con las derivaciones de la gramática G, y por lo tanto una cadena w estará en L(G) si y sólo si existe un camino en el grafo desde el vértice de S hasta el vértice de w (existen distintos algoritmos para determinar si existe un camino entre dos vértices de un grafo).

Por otra parte, se puede probar por diagonalización que los lenguajes sensibles al contexto no son todos los lenguajes recursivos. Al igual que las MT, las gramáticas pueden codificarse y ordenarse, en particular las gramáticas sensibles al contexto (queda como ejercicio proponer una codificación). Sea el lenguaje $L = \{w_i \mid w_i \notin L(G_i)\}$, tal que G_i es la gramática sensible al contexto i-ésima según el orden canónico:

- Se cumple que L es recursivo: dada una cadena w_i, se determina i según el orden canónico, se genera el código de la gramática 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: suponiendo que lo es, es decir que L se genera por una gramática G_k sensible al contexto, se llega a una contradicción. Si $w_k \in L$, entonces $w_k \in L(G_k)$, pero por la definición de L se cumple que $w_k \notin L$. Y si $w_k \notin L$, entonces $w_k \notin L(G_k)$, pero por la definición de L se cumple que $w_k \in L$.

TEMA 5.3. MÁQUINAS DE TURING RESTRINGIDAS

Los lenguajes regulares, libres de contexto y sensibles al contexto, se pueden reconocer por MT restringidas en lo que hace a su poder computacional comparado con el de las MT sin restricciones. Las analizamos a continuación con cierto detalle.

Un *autómata finito* (abreviado con AF) es una MT que tiene una sola cinta, la cinta de entrada, que es de sólo lectura, y sobre la cual el cabezal se mueve solamente a la derecha desde el primer símbolo de la entrada. Cuando el cabezal lee un blanco se detiene (acepta si y sólo si se detiene en un estado final). De este modo, los AF no tienen memoria, salvo la que pueden proveer los estados. No existe un alfabeto Γ dado que la única cinta es de sólo lectura. Por lo tanto, la función de transición δ está compuesta por ternas del conjunto $Q \times \Sigma \times Q$.

El autómata finito constituye un tipo de algoritmo ampliamente utilizado. Dos aplicaciones clásicas se relacionan con la implementación del analizador lexicográfico de los compiladores, y la verificación automática de programas utilizando modelos de transición de estados y lógica temporal (lo que se conoce como *model checking*).

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.

Ejemplo 5.2. Autómata finito

Sea el AF $M = <Q, \Sigma, \delta, q_0, F>$ siguiente:

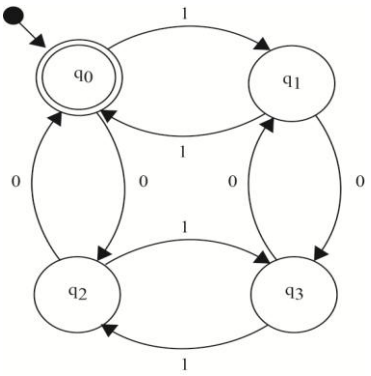
- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- $q_0 = q_0$
- $F = \{q_0\}$
- La función de transición δ se define de la siguiente manera:

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

Se cumple que L(M) es el conjunto de todas las cadenas de {0, 1}* que tienen un número par de unos y un número par de ceros. La prueba queda como ejercicio.

Fin de Ejemplo

Los autómatas finitos se pueden representar gráficamente mediante *diagramas de transición de estados*, como el siguiente, que corresponde al ejemplo anterior:



Los nodos representan los estados del autómata. Existe un arco orientado con nombre x del nodo q al nodo p, si está definido $\delta(q, x) = p$. El nodo que representa el estado inicial se señala por una flecha, y los estados finales se representan con nodos con contorno doble. Otra representación muy conocida de los lenguajes regulares la constituyen las *expresiones regulares*, que se utilizan a menudo para el reconocimiento de patrones en textos. Las expresiones regulares sobre un alfabeto Σ se definen inductivamente de la siguiente manera:

1. $\varnothing$ es una expresión regular que denota el lenguaje vacío.
2. λ es una expresión regular que denota el lenguaje $\{\lambda\}$, siendo λ la cadena vacía.
3. Si $x \in \Sigma$, entonces x es una expresión regular que denota el lenguaje $\{x\}$.
4. Si r y s son expresiones regulares que denotan los lenguajes R y S, respectivamente, entonces (r + s), (rs) y (r*) son expresiones regulares que denotan los lenguajes ($R \cup S$), ($R \cdot S$) y R*, respectivamente. El lenguaje R* se denomina *clausura* de R, y se define como la unión infinita de los lenguajes R^i, siendo $R^0 = \{\lambda\}$ y $R^i = R \cdot R^{i-1}$.

Por ejemplo, la expresión (0 + 1)* denota el lenguaje de las cadenas de ceros y unos, con cero o más dígitos. Y la expresión (0 + 1)*00(0 + 1)* denota el lenguaje de las cadenas de ceros y unos con al menos dos ceros consecutivos. Queda como ejercicio definir la expresión regular que representa el lenguaje reconocido por el autómata finito del ejemplo anterior.

Se prueba que la clase de los lenguajes regulares es cerrada con respecto a las operaciones de unión, intersección, complemento, concatenación y clausura. Se cumple además que la clase de los lenguajes regulares es la más pequeña que contiene a todos los conjuntos finitos y es cerrada con respecto a la unión, la concatenación y la clausura.

Al igual que en el caso de las MT generales, el modelo de los AF determinísticos es equivalente al de los AF no determinísticos. Como contrapartida, a diferencia de las MT generales, ahora con AF problemas como ¿$L(M) = \varnothing$?, ¿$L(M) = \Sigma^*$? y ¿$L(M) = L(M')$? son decidibles.

El segundo tipo de MT restringida que vamos a describir es el *autómata con pila* (abreviado con AP). Un AP es una MT no determinística con las siguientes restricciones:

- Tiene una cinta de entrada de sólo lectura y una cinta de trabajo que se comporta como una pila. De este modo, los AP tienen memoria, pero limitada en comparación con las MT generales.
- Hay dos tipos de pasos en un AP. En el primer caso se lee el estado corriente, el símbolo corriente de la entrada, y el símbolo tope de la pila, y como consecuencia el estado puede cambiar, el cabezal de la cinta de entrada se mueve un lugar a la derecha, y en la pila se puede reemplazar el símbolo tope por una

cadena, que puede ser vacía. El otro tipo de paso es como el anterior, salvo que en este caso no se lee la entrada ni se avanza sobre ella (por eso se lo conoce como *λ-movimiento*).

Un AP acepta una entrada cuando se vacía la pila. Una definición alternativa de aceptación es por la detención en un estado final.
El autómata con pila también constituye un tipo de algoritmo ampliamente utilizado. Un uso muy conocido se relaciona con la implementación del analizador sintáctico de los compiladores.
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.

Ejemplo 5.3. Autómata con pila

Sea el AP M = <Q, Σ, Γ, δ, q_o, Z, ∅> siguiente (utilizamos δ en lugar de Δ porque el AP planteado es determinístico, Z denota el primer símbolo de la pila, y ∅ significa que se acepta por pila vacía):

- Q = {q_a, q_b}
- Σ = {a, b}
- Γ = {Z, A, B}
- q_0 = q_a
- Z = Z
- La función de transición δ se define de la siguiente manera:

  1. δ(q_a, a, Z) = (q_a, AZ)
  2. δ(q_a, a, A) = (q_a, AA)
  3. δ(q_a, b, A) = (q_b, λ)
  4. δ(q_b, b, A) = (q_b, λ)
  5. δ(q_b, λ, Z) = (q_b, λ)

Se cumple que L(M) es el conjunto de las cadenas a^nb^n, con $n \geq 1$. La prueba queda como ejercicio.

Fin de Ejemplo

Se demuestra que la clase de los lenguajes libres de contexto es cerrada con respecto a las operaciones de unión, concatenación y clausura, mientras que no lo es con respecto a la intersección ni al complemento. A diferencia de las MT generales, en este caso el modelo de los AP no determinísticos no es equivalente al modelo de los AP determinísticos (es de destacar que la gran mayoría de los lenguajes de programación de uso general en la industria son libres de contexto y determinísticos). Otra diferencia con las MT generales es que con AP, problemas como ¿$w \in L(M)$? y ¿$L(M) = \varnothing$? son decidibles.

Por último, las MT restringidas con poder computacional para reconocer todos los lenguajes sensibles al contexto y sólo ellos se denominan *autómatas acotados linealmente* (el término se abrevia con AAL). Son MT no determinísticas con una sola cinta, sobre la que el cabezal nunca se mueve más allá de los extremos de la entrada. Por ejemplo, la MT del Ejemplo 1.1 que reconoce el lenguaje $L = \{a^nb^n \mid n \geq 1\}$ (como el autómata con pila visto recién) es un AAL determinístico.

La clase de los lenguajes sensibles al contexto es cerrada con respecto a las operaciones de unión, intersección, concatenación y clausura. Como sucede con las MT generales, también con AAL problemas como ¿$L(M) = \varnothing$?, ¿$L(M) = \Sigma^*$? y ¿$L(M) = L(M')$? son indecidibles.

TEMA 5.4. MÁQUINAS DE TURING CON ORÁCULO

El modelo de ejecución de las MT con oráculo, en su alcance más general, tiene más poder computacional que el modelo de MT que venimos utilizando. Permite establecer relaciones entre lenguajes y clases de lenguajes, tomando en consideración distintos supuestos (incluso falsos, como la existencia de una MT que reconoce HP y se detiene siempre, una MT que reconoce $L_U{}^C$, etc).

Dado un lenguaje A, una *máquina de Turing con oráculo A*, que se denota con M^A, es una MT que:

- Incluye dos cintas especiales, una *cinta de pregunta* y una *cinta de respuesta*.

- Incluye un estado especial *de pregunta*, identificado con $q_?$.
- Trabaja de la forma habitual, pero toda vez que su estado es $q_?$, si el contenido de su cinta de pregunta es una cadena v, la máquina escribe en un solo paso en su cinta de respuesta un 1 si $v \in A$ (respuesta positiva del oráculo), o un 0 si $v \notin A$ (respuesta negativa del oráculo).

Existe una definición equivalente que reemplaza la cinta de respuesta por dos estados especiales, uno para las respuestas positivas del oráculo y otro para las respuestas negativas. La MT M^A puede interpretarse como una MT M que invoca cero o más veces a una subrutina que reconoce el lenguaje A, teniendo en cuenta que no se establece *a priori* ninguna restricción sobre el lenguaje (en particular, podría no ser recursivamente numerable). Un lenguaje es *recursivamente numerable con respecto al oráculo A* si es reconocido por una MT M^A, y se denota como de costumbre con $L(M^A)$. En particular, si M^A se detiene a partir de todas las entradas, $L(M^A)$ es *recursivo con respecto al oráculo A*. Dos oráculos son *recursivamente equivalentes* si cada uno es recursivo con respecto al otro.

Ejemplo 5.4. Máquina de Turing con oráculo

La siguiente MT M^{HP} reconoce el lenguaje HP y se detiene siempre. Dada una entrada w, M^{HP} hace:

1. Copia w en su cinta de pregunta.
2. Pasa al estado $q_?$.
3. Si el oráculo responde que sí (no), entonces acepta (rechaza).

Fin de Ejemplo

Como se observa en el ejemplo, si no se restringe el tipo de oráculo utilizado se pueden contradecir enunciados de la jerarquía de la computabilidad. Notar que con el mismo oráculo anterior se puede reconocer HP^C, que no es recursivamente numerable. Naturalmente siguen habiendo más lenguajes que lenguajes reconocidos por MT, con o sin oráculos.

Como anticipamos en la introducción, las MT con oráculo son útiles para establecer relaciones entre lenguajes (en esta clase nos enfocaremos en la jerarquía de la

computabilidad, luego las utilizaremos en el marco de la complejidad computacional temporal). Por ejemplo, dados los oráculos

$A_1 = \{<M> \mid L(M) = \varnothing\}$

$A_2 = \{<M> \mid L(M^{A1}) = \varnothing\}$

se prueba que L_U es recursivamente equivalente a A_1, y que L_{Σ^*} es recursivamente equivalente a A_2. Esto indica de alguna manera que L_{Σ^*} es más difícil que L_U y A_1 (o lo que es lo mismo, $L_\varnothing$). Ya destacamos estas diferencias previamente, cuando probamos que L_{Σ^*} no pertenece al conjunto RE $\cup$ CO-RE, a diferencia de L_U y $L_\varnothing$. Si esta relación no es tan relevante cuando se trata con problemas indecidibles, su importancia crece cuando los problemas se consideran en términos de MT restringidas. Por ejemplo, con autómatas con pila M, los problemas ¿w $\in$ L(M)? y ¿L(M) = $\varnothing$? son decidibles, mientras que el problema ¿L(M) = Σ^*? no lo es. Y con autómatas finitos, con los que los tres problemas son decidibles, los dos primeros se resuelven en tiempo polinomial, y el último en tiempo exponencial.

Como segundo ejemplo de construcción de MT con oráculo, probamos a continuación uno de los enunciados recién formulados: el oráculo L_U es recursivamente equivalente al oráculo $L_\varnothing$, o en otras palabras, existe una MT con oráculo $L_\varnothing$ que acepta L_U y se detiene siempre, y existe una MT con oráculo L_U que acepta $L_\varnothing$ y se detiene siempre.

Ejemplo 5.5. Los oráculos L_U y $L_\varnothing$ son recursivamente equivalentes

Primero demostramos que si A = $L_\varnothing$, entonces existe una MT M^A tal que $L_U = L(M^A)$ y M^A se detiene siempre. Dada una entrada v, M^A hace:

1. Si v no es un par válido (<M>, w), rechaza.
2. Escribe en su cinta de pregunta un código $<M_w>$, tal que $L(M_w) = \Sigma^*$ si M acepta w, y $L(M_w) = \varnothing$ si M no acepta w (ver la construcción de $<M_w>$ en el Ejemplo 4.4).
3. Pasa al estado $q_?$. Si el oráculo responde que sí (no), entonces rechaza (acepta).

Claramente, M^A reconoce L_U y se detiene siempre (la prueba queda como ejercicio). Ahora probamos que si $B = L_U$, entonces existe una MT M^B tal que $L_\varnothing = L(M^B)$ y M^B se detiene siempre. Dado una entrada w, la M^B hace:

1. Si w no es un código válido <M>, acepta (seguimos con la convención de que se cumple $L(M) = \varnothing$ cuando <M> no es válido).
2. Genera un código <M'>, tal que M' ignora su entrada y simula, a partir de <M>, una MT $M_\varnothing{}^C$ que reconoce $L_\varnothing{}^C$ (ver la construcción de $M_\varnothing{}^C$ inmediatamente después del Ejemplo 4.10). De esta manera, $L(M') = \Sigma^*$ si $L(M) \neq \varnothing$, y $L(M') = \varnothing$ si $L(M) = \varnothing$.
3. Escribe en su cinta de pregunta el par (<M'>, λ) y pasa al estado $q_?$ Si el oráculo responde que sí (no), entonces rechaza (acepta).

Claramente, M^B reconoce $L_\varnothing$ y se detiene siempre (la prueba queda como ejercicio).

Fin de Ejemplo

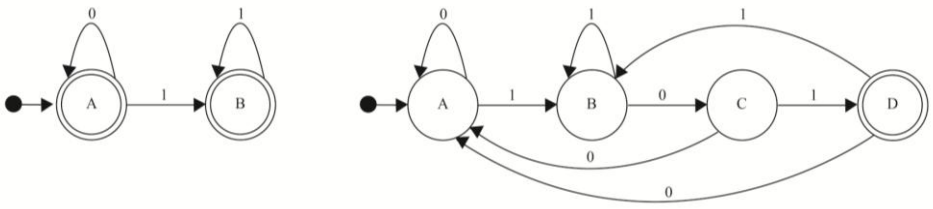
Si L_1 es recursivo con respecto a L_2, también se dice que existe una *Turing-reducción* de L_1 a L_2, y se denota con la expresión $L_1 \leq^T L_2$. De esta manera, podemos considerar a las Turing-reducciones como reducciones más generales que las que vimos hasta ahora (denominadas *m-reducciones* o reducciones *many-one* (muchos a uno) porque en general no son inyectivas). Claramente, si existe una m-reducción de L_1 a L_2, también existe una Turing-reducción entre ambos lenguajes, mientras que la recíproca no tiene por qué cumplirse (la prueba queda como ejercicio).

Dados dos lenguajes L_1 y L_2, se puede demostrar $L_2 \notin R$ probando $L_1 \leq^T L_2$ y $L_1 \notin R$, tal como se hace con las m-reducciones. En efecto, si L_2 fuera recursivo, entonces la MT M^{L2} que reconoce L_1 y se detiene siempre podría simularse por otra MT M' equivalente sin oráculo que se detiene siempre, por lo que L_1 sería recursivo (absurdo). Por lo tanto, las Turing-reducciones constituyen otra herramienta para probar no recursividad, muy útil cuando no existen o cuesta encontrar m-reducciones para el mismo objetivo. Como contrapartida, a diferencia de las m-reducciones, las Turing-reducciones no sirven para probar $L_2 \notin RE$ demostrando $L_1 \leq^T L_2$ y $L_1 \notin RE$: claramente se cumple para todo lenguaje L que $L \leq^T L^C$ (por ejemplo $HP^C \leq^T HP$, con $HP^C \notin RE$ y $HP \in RE$). Dicho contraste es un ejemplo de que cuanto más restrictivas son las reducciones utilizadas,

más propiedades y relaciones se pueden probar. Profundizaremos sobre este tema en la segunda parte del libro, cuando tratemos con reducciones acotadas temporal y espacialmente.

Ejercicios de la Clase 5

1. Completar la prueba del Teorema 5.1.
2. Construir una MT distinta a la del Teorema 5.1 para generar un lenguaje recursivamente numerable, de modo tal que no repita cadenas.
3. Proponer una codificación para las gramáticas, y construir una MT que determine, dado el código de una gramática G, si G es de tipo 1 (o sensible al contexto).
4. Establecer las diferencias entre las MT restringidas descriptas en la Clase 5 y las MT generales, y argumentar en cada caso la razón por la que las MT generales poseen mayor poder computacional. Hacer también un análisis comparativo considerando las MT restringidas entre sí.
5. Completar la prueba del Ejemplo 5.2. Definir además una expresión regular que represente el lenguaje reconocido por el autómata finito del ejemplo.
6. Describir los lenguajes representados por los siguientes diagramas de transición de estados:



7. Construir autómatas finitos que reconozcan los siguientes lenguajes. Representar además los lenguajes mediante expresiones regulares:
   i. El lenguaje de las palabras clave if, then, else, fi, while, do, od.
   ii. El lenguaje de las cadenas de {0, 1}* tales que a todo cero le sigue un uno.
   iii. El lenguaje de las cadenas de {0,1}* con tres ceros consecutivos.
8. Completar la prueba del Ejemplo 5.3.
9. Construir autómatas con pila que reconozcan los siguientes lenguajes. Determinar además si los lenguajes pueden reconocerse mediante autómatas finitos:
   i. El lenguaje de las cadenas de {0, 1}* con igual cantidad de unos y ceros.

ii. El lenguaje $L = \{0^n 1^m \mid 1 \leq n < m\}$.

iii. El lenguaje $L = \{a^n c b^n \mid n \geq 0\}$.

10. Probar la equivalencia entre el modelo de las MT con oráculo que utilizan una cinta de respuesta, y el modelo de las MT con oráculo que utilizan en cambio dos estados de respuesta, uno para las respuestas positivas y otro para las respuestas negativas.
11. Completar la prueba del Ejemplo 5.5.
12. Probar que si existe una m-reducción de L_1 a L_2, entonces también existe una Turing-reducción entre ambos lenguajes, mientras que la recíproca no tiene por qué cumplirse.
13. Probar que si existe una m-reducción de L_1 a L_2, y L_2 es recursivo (recursivamente numerable) con respecto a L_3, entonces L_1 es recursivo (recursivamente numerable) con respecto a L_3.
14. Dado $A_1 = \{<M> \mid L(M) = \varnothing\}$, probar que $L_U{}^C \leq^T A_1$, mientras que no se cumple $L_U{}^C \alpha A_1$.
15. Dado $A_2 = \{<M> \mid L(M^{A1}) = \varnothing\}$, siendo A_1 el oráculo definido en el ejercicio anterior, probar:

    i. $L = \{<M> \mid L(M) = \Sigma^*\}$ es recursivamente equivalente a A_2.

    ii. $L = \{<M> \mid L(M)$ es finito$\}$ es recursivamente equivalente a A_2.

Notas y bibliografía para la Parte 1

Las máquinas de Turing fueron introducidas por A. Turing en (Turing, 1936). En este trabajo se define el concepto de número computable, y se presenta en términos de dichas máquinas un modelo general de computación, equivalente al λ-cálculo (Church, 1936), las máquinas de Post (Post, 1936), las funciones recursivas parciales (Kleene, 1936), y los algoritmos de Markov (Markov, 1954), entre otros modelos. Turing probó además la equivalencia de su formalismo con el de A. Church, y que el problema de la detención de una máquina de Turing es indecidible (Church hizo lo propio con el problema de la equivalencia de dos expresiones del λ-cálculo). Por aquel entonces, Turing acababa de terminar sus estudios de grado en matemáticas, y luego de su trabajo fue invitado por Church a Princeton para doctorarse bajo su tutela.

El formalismo de E. Post es muy similar al de Turing. Siendo profesor en Nueva York envió a publicar el manuscrito que lo describía un poco después que lo hiciera Turing. Post fue también autor de otro modelo computacional equivalente, los sistemas que llevan su nombre, basados en reglas de reescritura o producciones como los algoritmos de Markov.

Por su parte, S. Kleene demostró la equivalencia de su formalismo con el λ-cálculo de Church. A él se le debe el enunciado de la Tesis de Church-Turing, de 1952. El modelo de Kleene, basado en las funciones recursivas parciales, incluye a las funciones recursivas primitivas con las que había trabajado K. Gödel previamente (enseguida hacemos referencia a este matemático).

Una de las motivaciones más importantes de Turing era resolver la cuestión de la decidibilidad o indecidibilidad del *Entscheidungsproblem*, el problema consistente en determinar si una fórmula de la lógica de primer orden es un teorema. Este problema se formulaba en el marco de un ambicioso proyecto de los matemáticos formalistas. D. Hilbert, uno de los más grandes matemáticos del siglo pasado, había planteado un plan para mecanizar las pruebas matemáticas. Una respuesta positiva y admirable en este sentido fueron los Principia Mathematica de B. Russell y A. Whitehead, en 1913. Otros resultados positivos, como la demostración de la completitud de la lógica de primer orden por parte de Gödel, alumno de Hilbert, en 1929, siguieron alentando a los formalistas. Pero fue el mismo Gödel quien en (Gödel, 1931) acabó abruptamente con el proyecto: con su famoso Teorema de Incompletitud demostró que todo sistema

axiomático recursivo y consistente que contenga "suficiente" aritmética (suma y multiplicación) tiene enunciados indecidibles (en particular la propia consistencia del sistema). Reforzando los resultados negativos, Turing y Church demostraron en 1936, de manera independiente y con distintos métodos, la indecibilidad del Entscheidungsproblem. Ambos se enfocaron en una instancia particular de la lógica de primer orden, la *lógica canónica de primer orden* F_0, también llamada *Engere Prädikatenkalkül* por Hilbert, y *cálculo funcional puro de primer orden* por Church, teniendo en cuenta el resultado que establece que si F_0 es decidible, entonces también lo es cualquier lógica de primer orden.

El concepto de máquina no determinística surgió como necesidad más de la complejidad computacional que de la computabilidad. Fue introducido a fines de la década de 1950. Ver por ejemplo (Rabin & Scott, 1959).

Varios ejemplos de reducciones de problemas desarrollados en la Clase 4 provienen de (Hopcroft & Ullman, 1979). La indecidibilidad del problema de correspondencia de Post se demuestra en (Post, 1946), y la reducción presentada en este libro, que a partir de dicho problema prueba la indecidibilidad de la lógica de primer orden, pertenece a Church y se describe por ejemplo en (Huth & Ryan, 2004). En la misma clase hemos estudiado el Teorema de Rice para caracterizar a los lenguaje recursivos. H. Rice estableció también una caracterización de los lenguajes recursivamente numerables. Su trabajo completo aparece en (Rice, 1953) y (Rice, 1956).

El modelo de las máquinas RAM, también equivalente al de las máquinas de Turing, se introdujo en (Shepherdson & Sturgis, 1963). Se considera que (McCulloch & Pitts, 1943) es la primera conceptualización de los autómatas finitos. La jerarquía de Chomsky asociada a los tres tipos de máquinas de Turing restringidas presentadas en la Clase 5 (autómatas finitos, autómatas con pila y autómatas acotados linealmente), se publicó en (Chomsky, 1956). Turing introdujo las máquinas con oráculo en (Turing, 1939), para estudiar los sistemas lógicos basados en ordinales.

Para profundizar en los distintos temas sobre computabilidad considerados en las cinco clases de la primera parte del libro, incluyendo referencias históricas y bibliográficas, se recomienda recurrir a (Hopcroft & Ullman, 1979) y (Lewis & Papadimitriou, 1981). Sugerimos también para consulta (Davis, 1958), (Minsky, 1967), (Rogers, 1987) y (Sipser, 1997). Otras dos lecturas recomendadas son las siguientes. En (Penrose, 1996), a propósito de cuestionar a quienes sostienen que las computadoras igualarán e incluso superarán a la mente humana (corriente que se conoce como *inteligencia artifical*

fuerte), el autor recorre en los capítulos 2, 3 y 4 temas tales como la Tesis de Church-Turing, el λ-cálculo, el Teorema de Incompletitud de Gödel, ejemplos de problemas indecidibles y la teoría de la complejidad. Por su parte, (Herken, 1994) recopila numerosos artículos de importantes referentes de la teoría de la computación, elaborados cuando se cumplieron cincuenta años de la publicación de Turing que introdujo las máquinas que llevan su nombre; los cinco trabajos de la primera parte de esta recopilación tratan el aporte de Turing, y su relación con otras contribuciones (Church, Gödel, Hilbert, Kleene y Post) y con el desarrollo de la computación.

A lo largo de todo el libro hacemos distintas referencias a la lógica, en lo que hace a su relación con la computabilidad, la complejidad computacional y la verificación de programas. Entre otras obras introductorias a la lógica, recomendamos (Enderton, 1972), (Mendelson, 1979) y (Schoenfield, 1967). También sugerimos la lectura de los siguientes trabajos. (Huth & Ryan, 2004) está orientado a las carreras de informática, relacionando íntimamente la lógica con las herramientas existentes para la especificación y verificación del *hardware* y el *software*; además de la lógica proposicional y de primer orden, trata la lógica temporal, tanto lineal como ramificada, y técnicas formales de verificación teniendo en cuenta las distintas lógicas estudiadas. En los capítulos 4 a 6 de (Papadimitriou, 1994) se analizan, respectivamente, la lógica proposicional, la lógica de primer orden y la teoría de números, a partir del convencimiento del autor de que el poder expresivo de la lógica la convierte en un instrumento extremadamente valioso para el estudio de la complejidad computacional. Finalmente, (Martínez & Piñeiro, 2009) describe de una manera autocontenida el Teorema de Incompletitud de Gödel; en particular, en los últimos capítulos los autores se enfocan en la causa de la incompletitud a partir del poder expresivo del lenguaje de la aritmética cuando incluye la suma y la multiplicación.

Se detallan a continuación las referencias bibliográficas mencionadas previamente:

- Chomsky, N. (1956). "Three models for the description of language". *IRE Trans. on Information Theory*, 2(3), 113-124.
- Church, A. (1936). "An unsolvable problem of elementary number theory". *American Journal of Mathematics*, 58(2), 345-363.
- Davis, M. (1958). *Computability and unsolvability*. McGraw-Hill.
- Enderton, H. (1972). *A mathematical introduction to logic*. Academic Press.

- Gödel, K. (1931). "Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme, I". *Monatshefte fur Math, Und Physik*, 38, 173-198. Para leer una traducción al castellano, ver por ejemplo: García Suárez, A., Garrido, M. & Valdés Villanueva, L. (2006). "Sobre proposiciones formalmente indecidibles de los Principia Mathematica y sistemas afines". *KRK Ediciones*.
- Herken, R. (1994). *The universal Turing machine. A half-century survey, second edition*. Springer-Verlag.
- Hopcroft, J. & Ullman J. (1979). *Introduction to automata theory, languages, and computation*. Addison-Wesley.
- Huth, M. & Ryan, M. (2004). *Logic in computer science*. Cambridge University Press.
- Kleene, S. (1936). "General recursive functions of natural numbers". *Mathematische Annalen*, 112, 727-742.
- Lewis, H. & Papadimitriou, C. (1981). *Elements of the theory of computation*. Prentice-Hall.
- Markov, A. (1954). "The theory of algorithms". *Trudy Mat., Inst. Steklov, Acad. Sci. USSR*, 42, 3-375.
- Martínez, G. & Piñeiro, G. (2009). *Gödel ∀ (para todos)*. Seix Barral.
- McCulloch, W. & Pitts, E. (1943). "A logical calculus of the ideas immanent in nervous activity". *Bulletin Mathematical Biophysics*, 5, 115-133.
- Mendelson, E. (1979). *Introduction to mathematical logic, second edition*. van Nostrand.
- Minsky, M. (1967). *Computation: finite and infinite machines*. Prentice-Hall.
- Papadimitriou, C. (1994). *Computational complexity*. Addison-Wesley.
- Penrose, R. (1996). *La mente nueva del emperador. En torno a la cibernética, la mente y las leyes de la física*. Consejo Nacional de Ciencia y Tecnología, Fondo de Cultura Económica, México.
- Post, E. (1936). "Finite combinatory process – Formulation 1". *Journal of Symbolic Logic*, 1(3), 103-105.
- Post, E. (1946). "A variant of a recursively unsolvable problem". *Bull. Amer. Math. Soc.*, 52(4), 264-268.

- Rabin, M. & Scott, D. (1959). “Finite automata and their decision problems”. *IBM Journal of Research and Development*, 3, 114-125.
- Rice, H. (1953). “Classes of recursively enumerable sets and their decision problems”. *Trans. AMS*, 89, 25-59.
- Rice, H. (1956). “On completely recursively enumerable classes and their key arrays”. *Journal of Symbolic Logic*, 21, 304-341.
- Rogers, H. (1987). *Theory of recursive functions and effective computability, second edition*. MIT Press.
- Shepherdson, J. & Sturgis, H. (1963). “Computability of recursive functions”. *Journal of ACM*, 10, 217-255.
- Schoenfield, J. (1967). *Mathematical logic*. Addison-Wesley.
- Sipser, M. (1997). *Introduction to the theory of computation*. PWS Publishing Company.
- Turing, A. (1936). “On computable numbers, with an application to the Entscheidungsproblem”. *Proc. London Math. Society*, 2(42), 230-265.
- Turing, A. (1939). “Systems of logic Based on ordinals”. *Proc. London Mathematical Society*, 2(45), 161-228.

Parte 2. Complejidad computacional

Las cinco clases de la segunda parte del libro están dedicadas al análisis de la *complejidad computacional* de los problemas decidibles, medida en términos del tiempo y el espacio consumidos por una máquina de Turing, que se sigue utilizando como modelo de ejecución (se mantiene el modelo estándar determinístico de varias cintas y dos estados finales). En realidad analizamos en profundidad sólo la complejidad temporal; la complejidad espacial no entra en el alcance del libro, aunque de todos modos le dedicamos toda una sección en la Clase 9.

En la Clase 6 se describe la *jerarquía de la complejidad temporal*, que establece cómo se distribuyen los problemas decidibles entre distintas clases de acuerdo a su complejidad calculada por la cantidad de pasos de las máquinas de Turing que los resuelven. Se introducen las definiciones más relevantes, incluyendo cómo representar "razonablemente" datos y problemas. Se presentan las clases de problemas P y NP, que son las que incluyen a los problemas de tiempo de resolución polinomial determinístico y no determinístico, respectivamente. Se fundamenta la convención que establece que los problemas de P son los que tienen resolución eficiente. Se destaca la importancia de la clase NP por incluir a gran parte de los problemas de interés computacional. Y se plantea la pregunta abierta hasta el día de hoy de si P ≠ NP. También se formulan y demuestran propiedades generales de la jerarquía temporal, sin particularizar en P y NP, como el Teorema de Aceleración Lineal (*Linear Speed Up Theorem*), y un par de enunciados relacionados con la densidad de la jerarquía temporal (siempre se puede definir una clase de problemas mayor, dentro de R).

En la Clase 7 se describen *las clases P y NP*. Se muestran ejemplos de problemas en P y NP, y se formulan y demuestran propiedades de las dos clases. En particular, se caracteriza a los problemas de NP como aquéllos cuyas soluciones se pueden verificar en tiempo eficiente (noción de certificado suscinto), prescindiéndose de este modo de utilizar máquinas de Turing no determinísticas en la definición, lo que facilita la comprensión de lo que significa la pertenencia a esta clase de complejidad. Asociada a la conjetura P ≠ NP, se plantea la oposición entre los algoritmos sofisticados que caracterizan a la clase P y los algoritmos de "fuerza bruta" que caracterizan a la clase NP. Una última caracterización establece la correspondencia entre los problemas de NP y los que pueden especificarse mediante la lógica existencial de segundo orden.

El tema central de la Clase 8 lo constituyen los *problemas NP-completos* (clase NPC), los más difíciles de la clase NP en el sentido de la complejidad temporal, en la práctica "condenados" a no estar en P salvo que se cumpla P = NP. Se retoman las reducciones de problemas, ahora acotadas a un tiempo de ejecución determinístico polinomial, para encontrar sistemáticamente problemas NP-completos (el mecanismo es similar al utilizado en la Parte 1 del libro para poblar las distintas clases de la jerarquía de la computabilidad). Mediante un teorema conocido como el Teorema de Cook, se introduce un primer problema NP-completo, el problema SAT, que consiste en determinar si una fórmula booleana es satisfactible. A partir del mismo se prueba luego la NP-completitud de otros problemas, relacionados con la lógica y la teoría de grafos. Se destacan dos propiedades de todos los problemas NP-completos conocidos: son p-isomorfos y densos.

En la Clase 9 se profundiza en la jerarquía temporal, presentándose *otras clases de complejidad* que completan el "mapa". Se introducen los problemas de complejidad intermedia entre P y NPC dentro de NP (la clase NPI), y los problemas tales que sus complementos están en NP (la clase CO-NP). Los problemas de CO-NP se identifican con los que se pueden especificar mediante la lógica universal de segundo orden. También se muestran ejemplos de problemas con tiempo mínimo de resolución efectivamente exponencial. La última sección está dedicada a la complejidad espacial. Se introducen las definiciones más relevantes; se desarrollan ejemplos; se presenta la jerarquía espacial, y se la relaciona con la jerarquía temporal para establecer una única jerarquía espacio-temporal; se introducen las reducciones logarítmicas en espacio para poblar las distintas clases de la jerarquía espacial (en realidad de la jerarquía espacio-temporal, porque se demuestra fácilmente que una reducción logarítmica en espacio es polinomial en tiempo); y se trata la completitud de los problemas, ahora de cualquier clase de complejidad.

La Clase 10 es una clase de *miscélaneas de complejidad temporal*. Primeramente, después de haberse puesto foco casi exclusivamente en los problemas de decisión a lo largo de las nueve clases anteriores, se tratan los problemas más generales, conocidos como problemas de búsqueda, a partir de los cuales se deben ahora obtener efectivamente soluciones, no solamente las respuestas "sí" o "no". En este contexto se introducen las Cook-reducciones y las Levin-reducciones. También se introducen las aproximaciones polinomiales, para obtener eficientemente óptimos (máximos o mínimos) con error acotado en los casos en que no pareciera factible la existencia de

algoritmos eficientes para resolver las optimizaciones sin limitar el tiempo de ejecución. Otra sección retoma el modelo de las máquinas de Turing con oráculo, en este caso acotadas a un tiempo de ejecución polinomial, con las que se puede profundizar en el análisis de la conjetura P ≠ NP (se "relativizan" relaciones entre P y NP, NP y CO-NP, etc., asumiendo que las máquinas de Turing tienen un oráculo determinado). Por medio de estas máquinas se presenta además otra jerarquía de complejidad, conocida como la jerarquía polinomial o PH, que comprende problemas que van desde P hasta PSPACE (esta última clase incluye a los problemas que se resuelven en espacio determinístico polinomial). Los problemas de PH se identifican con los que se pueden especificar mediante la lógica de segundo orden, ahora sin restricciones sintácticas. En una siguiente sección se describen las máquinas de Turing probabilísticas, cuyo criterio de aceptación es distinto al de las máquinas de Turing comunes. Se presenta una jerarquía de complejidad definida en términos de estas máquinas, y se la relaciona con la jerarquía temporal. La última sección está dedicada a la clase NC, la clase de los problemas de P que se consideran eficientemente paralelizables. Como novedad, para su estudio se introduce un modelo de ejecución que no se basa en las máquinas de Turing, los circuitos booleanos, muy tenidos en cuenta a la hora de encontrar cotas mínimas de tiempo.

Clase 6. Jerarquía de la complejidad temporal

Caracterizados los problemas decidibles en el marco de la jerarquía de la computabilidad, ahora se los va a estudiar desde el punto de vista de los recursos que requieren las máquinas de Turing para resolverlos. Nos referiremos fundamentalmente al *tiempo*, es decir a la cantidad de pasos efectuados por las MT. Sobre el *espacio*, es decir la cantidad de celdas utilizadas por las MT, haremos también un análisis más adelante pero mucho más acotado, dado que no se considera en el alcance de este libro.

A las medidas de complejidad de este tipo, relacionadas con las computaciones de las MT, se las conoce como *dinámicas*. Otra medida dinámica es la cantidad de veces que en la ejecución de una MT con una sola cinta, su cabezal cambia de dirección. También se puede encarar el estudio de una manera más abstracta, definiendo una axiomática relacionada con algún recurso genérico (M. Rabin formuló las bases de este enfoque en 1960, que años más tarde desarrolló M. Blum). El tópico de las medidas dinámicas de complejidad, de las que el tiempo y el espacio son las más populares e intuitivas, es lo que trata la *complejidad computacional*, expresión introducida por J. Hartmanis y R. Stearns en 1965.

Una aproximación diferente para el estudio de la complejidad se basa en las medidas *estáticas*, en cuyo caso se hace referencia a la *complejidad estructural* de los problemas. Un ejemplo de estas medidas de complejidad es el producto entre la cantidad de estados y el tamaño del alfabeto de una MT. Esta fue una de las primeras medidas de complejidad definidas, introducida por C. Shannon en 1956. Otro ejemplo es el análisis de los problemas en relación a la complejidad estructural de las MT que los resuelven, teniendo en cuenta la jerarquía de Chomsky (mencionada en el Tema 5.2).

En esta clase presentamos las definiciones básicas de la complejidad computacional temporal (o directamente complejidad temporal), y la jerarquía de clases de problemas asociada, conocida como *jerarquía de la complejidad temporal* (o directamente *jerarquía temporal*). Nuestro estudio se sigue centrando en los problemas de decisión, por lo que la mayoría de las veces continuamos empleando de manera indistinta los términos problema y lenguaje.

Definición 6.1. Conceptos básicos de la complejidad temporal

Como las instancias de un problema pueden ser de cualquier tamaño, y en general una MT que lo resuelve tarda más (efectúa más pasos) a medida que las entradas son más grandes, resulta natural definir el tiempo de ejecución en términos del tamaño de las entradas. Precisando, la idea es medir el tiempo de ejecución de una MT a partir de una entrada w, con |w| = n, por medio de una función T(n), y analizar la razón de crecimiento de la función temporal, categorizando la complejidad del problema asociado según si la MT que lo resuelve trabaja en tiempo lineal, polinomial, exponencial, doble exponencial, etc.

Se va a justificar enseguida por qué de acuerdo a nuestro enfoque, en las mediciones es irrelevante considerar factores constantes. En efecto, trabajamos con órdenes de magnitud. En lugar de funciones T se utilizan funciones O(T), que se leen "orden de T". La expresión O(T) denota el conjunto de todas las funciones f que cumplen $f(n) \leq c.T(n)$, para toda constante $c > 0$ y todo número natural $n \geq 0$.

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

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

Fin de Definición

De esta manera, si un problema pertenece a la clase DTIME(T(n)) (respectivamente NTIME(T(n))), cualquiera sea la instancia w considerada la respuesta de la MTD (respectivamente MTN) que lo resuelve no tarda más de O(T(|w|)) pasos. Esto significa que el criterio de medición es por el *peor caso*. Otro criterio es por el *caso promedio*, pero para ello se necesita conocer cómo se distribuyen las entradas. Por otro lado, que un problema pertenezca a una clase temporal no implica que no tenga un tiempo de resolución menor. El ideal por supuesto es determinar la cota temporal mínima,

establecer la complejidad intrínseca de un problema, independientemente de los algoritmos que se conozcan, pero este objetivo es difícil en general. Lo habitual es identificar la complejidad temporal de un problema con la del algoritmo encontrado más eficiente para resolverlo. Trataremos esta cuestión a lo largo de las próximas clases, con foco en clases específicas de problemas.

En la definición anterior aparecen las MTD y MTN con varias cintas. Otra cosa que vamos a justificar enseguida es por qué utilizaremos las MTD con varias cintas como modelo estándar para el análisis de la complejidad temporal. A propósito, cabe recordar que en la primera parte del libro (ver Ejemplo 1.5) se mostró que pasar de una MT con varias 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 demostrar también que reduciendo la cantidad de cintas a dos, el retardo disminuye a $O(n.\log_2 n)$. Veremos que estos retardos no invalidan la adopción del modelo de las MTD con varias cintas como MT estándar. El siguiente es un primer ejemplo de medición de tiempo, considerando MTD.

Ejemplo 6.1. Modelo de ejecución estándar

Sea nuevamente el lenguaje $L = \{w \mid w \in \{a, b\}^* \text{ y } w \text{ es un palíndrome}\}$ presentado en la primera parte del libro. Una posible MTD M, con una cinta, para reconocer L, es la que a partir de una entrada w trabaja de la siguiente manera:

1. Si w es la cadena vacía λ, acepta.
2. Lee el primer símbolo de w. Si no es ni *a* ni b, rechaza. En caso contrario lo elimina y se mueve a la derecha hasta encontrar el último símbolo de w. Si no existe, acepta. Si existe y es distinto del primero, rechaza.
3. Elimina el último símbolo, vuelve a la izquierda hasta encontrar, si existe, el nuevo primer símbolo de w, y comienza el ciclo otra vez a partir del paso 1.

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

En cambio, la MTD, con dos cintas, descripta en el Ejemplo 1.5, reconoce L en menos tiempo que M: lo hace en tiempo lineal. La repetimos a continuación. Dada una entrada w, hace:

1. Copia w de la cinta 1 a la cinta 2. Si detecta un símbolo distinto de *a* y de b, rechaza. Si no, queda apuntando al primer símbolo de w en la cinta 1 y al último símbolo de w en la cinta 2.
2. Compara los dos símbolos apuntados. Si son blancos, acepta. Si son distintos, rechaza. 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 el paso 2.

Esta MT hace unos 3n pasos. Por lo tanto, trabaja en tiempo T(n) = O(n), es decir que L $\in$ DTIME(n), lo que es más preciso que decir que L $\in$ DTIME(n^2).

Fin de Ejemplo

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

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

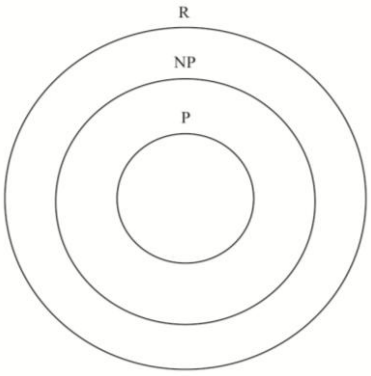
Notar que $O(n^i)$ reúne a todas las funciones polinomiales $T(n) = a_0 + a_1 n + \ldots + a_i n^i$. Por lo tanto, la clase P agrupa a los problemas que se resuelven en tiempo determinístico polinomial, y NP es la clase de los problemas que se resuelven en tiempo no determinístico polinomial.

Se cumple por definición que P $\subseteq$ NP, y además que las dos clases están incluidas estrictamente en R (al final de la clase se hace una referencia a cómo se puede probar esto).

En cambio, sólo podemos afirmar que "se sospecha" que P $\neq$ NP.

La conjetura P $\neq$ NP, luego de más de cuarenta años, aún no pudo probarse (ni refutarse), y no hay indicios de que la demostración se logre en el corto plazo. Constituye el problema abierto más importante de la complejidad computacional (tanto es así que en el año 2000, el prestigioso Clay Mathematics Institute lo catalogó como uno de los siete problemas matemáticos más importantes del milenio).

En P y NP se identifican numerosísimos e importantes problemas de la computación. Asumiendo P $\neq$ NP, la figura siguiente muestra una primera versión de la jerarquía temporal:



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 resoluciones que consumen más que tiempo determinístico polinomial no se consideran aceptables. La condición de que el tiempo sea determinístico es insoslayable (recordar la simulación determinística de una MTN que se mostró en el Ejemplo 1.7, la cual tarda tiempo exponencial). La convención de identificar al tiempo determinístico polinomial con lo tratable, aceptable o eficiente, no es arbitraria, se condice con la realidad. Podría argumentarse alguna inconsistencia en el caso de entradas pequeñas, en que $a.n^b$ puede superar ampliamente a $c^{d.n}$, dadas las constantes a, b, c, d (consideremos, por ejemplo, $30n^{20}$ vs $2^{0,1n}$). Pero estas relaciones no ocurren en la práctica. Además, la complejidad computacional trata con entradas de todos los tamaños, y las funciones polinomiales, cuando n tiende a infinito, se mantienen por debajo de las funciones exponenciales.

De esta manera, R – P queda como la clase de los problemas *intratables*, siendo entonces la frontera de la clase P la que separa los problemas con resolución temporal aceptable o eficiente de los de resolución temporal inaceptable o ineficiente.

Para simplificar la presentación, asumiremos que lo que no es polinomial es exponencial (en la Clase 9 hacemos alguna referencia a la jerarquía exponencial). En este contexto, identificaremos de ahora en más con EXP (por tiempo exponencial) a la clase de todos los problemas decidibles, es decir R.

Con las consideraciones previas se justifica el uso de las MTD, con cualquier cantidad de cintas, como modelo de ejecución "razonable" para el análisis de la complejidad temporal de los problemas. En efecto, un problema que se resuelve en tiempo polinomial con una MTD con K_1 cintas, también se resuelve en tiempo polinomial con una MTD con K_2 cintas, cualesquiera sean K_1 y K_2. En este sentido, otro modelo razonable lo constituyen las máquinas RAM. En cambio, el modelo de las MTN no es

razonable: podría darse la inconsistencia de que un problema con resolución exponencial mediante una MTD, tuviera resolución polinomial con una MTN. Que los tiempos en dos modelos razonables de ejecución se relacionan polinomialmente, se enuncia en la Tesis de Cobham-Edmonds. Esta tesis, que data de 1965, es una de las primeras referencias a P, identificándola como la clase de los problemas que se resuelven eficientemente.

Además de la elección de un modelo razonable de ejecución, también se debe considerar una representación razonable de los datos (las cadenas) tratados por las MT, porque de lo contrario se producirían inconsistencias como las que muestra el ejemplo siguiente.

Ejemplo 6.2. Representación estándar de los números

Sea L el lenguaje de los números primos. Y sea M una MT que lo reconoce de la siguiente manera (no óptima): dada una entrada w, M acepta w si y sólo si ninguno de los números 2, 3, …, w – 1, divide a w.

Por lo tanto, la MT M lleva a cabo O(w) iteraciones, es decir que la cantidad de iteraciones depende del número de entrada, independientemente de su representación. Veamos cómo influye la representación de los números en el tiempo de trabajo de M, para concluir que la representación estándar de los números no puede ser la notación unaria. Para simplificar, asumimos que t es el tiempo que consume cada división, independientemente de la representación.

- Si la entrada w se representa en notación unaria, entonces $|w| = w = n$, y por lo tanto M trabaja en tiempo O(n).t, es decir lineal en n.
- Si en cambio w se representa en notación binaria, entonces $|w| = \log_2 w = n$, y por lo tanto M trabaja en tiempo $O(2^n).t$, es decir exponencial en n. Naturalmente, lo mismo sucede para cualquier notación que no sea la unaria.

Fin de Ejemplo

Obviamente deben evitarse inconsistencias de este tipo. Si un problema es tratable (intratable) con una representación, también debe ser tratable (intratable) con otra. Esta es la noción de representación razonable. De ahora en más tendremos en cuenta representaciones con las siguientes características:

- Los números se representan en notación distinta de la unaria. Además de la inviabilidad práctica de la representación unaria para números muy grandes, puede suceder, como se vio en el último ejemplo, que un problema con resolución exponencial utilizando notación no unaria tenga resolución polinomial con notación unaria. En cambio, esto no ocurre con las notaciones no unarias, cualesquiera sean las bases.
- Se puede transformar eficientemente una representación razonable en otra. En particular, considerando números, 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 base *a* mide $\log_a w$, en base b mide $\log_b w$, y se cumple que $\log_a w / \log_b w = \log_a b$. De este modo, en general omitiremos especificar las bases de los logaritmos.
- La representación de los conjuntos, listas, etc., consiste en la secuencia de sus elementos, separados por símbolos apropiados.

Todavía hay un tercer parámetro de razonabilidad, en este caso relacionado con la representación de (las instancias de) los problemas, que se ejemplifica a continuación. Consideramos un ejemplo sobre grafos. En las próximas clases trataremos varios problemas con grafos. Lo que se muestra en el ejemplo, de todos modos, es extensible a todos los problemas.

Ejemplo 6.3. Representación estándar de los problemas

Sea L = {(G, v_1, v_2, K) | 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 establecer si un grafo tiene un camino de longitud al menos K entre dos vértices determinados.

Una representación habitual de un grafo G, que utilizaremos frecuentemente en las próximas clases, consiste en un par (V, E), donde V = {1, …, m} es el conjunto de vértices y E es el conjunto de arcos de G. Los vértices se representan en notación binaria y se separan por un símbolo #. Los arcos se representan por pares de números naturales (los vértices que los determinan) en notación binaria y se separan por un #, lo mismo que sus vértices. Finalmente, V y E se separan por dos # consecutivos. Por ejemplo, si G = ({1, 2, 3, 4}, {(2, 4), (4, 1)}), la representación de G es 1#10#11#100##10#100#100#1. Por convención denotaremos siempre con m al tamaño de V, y asumiremos que G es un grafo no orientado, salvo mención en contrario.

Ni con ésta, ni con otras representaciones razonables conocidas, como la matriz de adyacencia por ejemplo, se conoce resolución eficiente para L. En cambio, si un grafo se representa enumerando todos sus caminos, una resolución eficiente trivial es la siguiente: se recorren uno por uno los caminos hasta encontrar eventualmente un camino entre el vértice v_1 y el vértice v_2 de longitud al menos K.

Esta última representación no es razonable, la complejidad temporal del problema está oculta en su representación. Además, el tiempo de transformación de cualquiera de las representaciones razonables conocidas a ésta es exponencial.

Fin de Ejemplo

Las clases P y NP se tratan en detalle a partir de la clase siguiente. En lo que sigue presentamos algunas características 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, destacamos que si bien $T_1(n) = O(T_2(n))$ implica que $DTIME(T_1(n)) \subseteq DTIME(T_2(n))$ (la prueba queda como ejercicio), esta inclusión no es necesariamente estricta. En efecto, por ejemplo el salto de una clase a otra que la incluya estrictamente debe ser mayor que lo determinado por factores constantes. En este caso, la justificación es que las diferencias por factores constantes se deben a características de las MT utilizadas, como el tamaño de los alfabetos, que a partir de construcciones adecuadas se pueden eliminar.

Más precisamente, el Teorema de Aceleración Lineal (*Linear Speed Up Theorem*), en una de sus variantes, establece que si existe una MT M_1 con K cintas que trabaja en tiempo T(n), entonces existe una MT M_2 con K + 1 cintas equivalente que trabaja en tiempo c.T(n) + (n + 1), para c = 1/2, 1/4, 1/8, …, es decir $1/2^d$, siendo d cualquier número natural mayor que cero (para simplificar la notación no utilizamos el operador de parte entera, pero la expresión se debe entender considerándolo; esto es aplicable a todo lo que sigue, en ésta y las próximas clases). En otras palabras, como ya dijimos, una clase $DTIME(T_2(n))$ no incluye más problemas que una clase DTIME $(T_1(n))$, si es que $T_2(n)$ difiere de $T_1(n)$ en un factor constante (tener en cuenta que el sumando n + 1 es lo mínimo que puede valer una función temporal T(n)).

No probaremos el teorema. La idea general de su demostración es la siguiente:

- Se construye M_2 a partir de M_1, de modo tal que el comportamiento sea el mismo, y que el alfabeto de M_2 incluya todas las secuencias de r símbolos del alfabeto de M_1, para un r determinado que depende de c. El propósito es que M_2 haga en un paso lo que a M_1 le lleva varios, definiendo una función de transición apropiada.
- M_2 debe primero comprimir su entrada. Al tener una cinta más que M_1, la compresión no le lleva más que lo que tarda la lectura de su entrada. Con una sola cinta la compresión le llevaría $O(n^2)$ pasos (de todos modos, si M_1 tuviera más de una cinta, M_2 no necesitaría una cinta más que M_1, porque luego de la compresión podría utilizar su cinta de entrada como una cinta de trabajo).

Por ejemplo, dada la cadena 3726886273, determinar si es un palíndrome reconociendo primero el 3 de la izquierda, luego el 3 de la derecha, luego el 7 de la izquierda, etc., tarda aproximadamente el doble, si no se considera el tiempo de compresión inicial, que procesar la cadena de a pares, reconociendo primero el 37 de la izquierda, luego el 73 de la derecha, luego el 26 de la izquierda, etc.

De esta manera hemos justificado lo enunciado al comienzo de la clase, que en las mediciones es irrelevante considerar factores constantes.

Para que una clase DTIME($T_2(n)$) incluya estrictamente a una clase DTIME($T_1(n)$), no alcanza ni siquiera con que $T_2(n) > T_1(n)$ (ignorando factores constantes), como se demostrará enseguida. Antes es oportuno destacar que en las definiciones de la jerarquía temporal se necesita trabajar con funciones temporales de "buen comportamiento", denominadas *tiempo-construibles*. Una función $T: N \rightarrow N$ es tiempo-construible si para toda entrada w, con $|w| = n$, existe una MT que trabaja en tiempo determinístico exactamente T(n). Claramente, las funciones tiempo-construibles son totales y computables, mientras que la recíproca no tiene por qué cumplirse (la prueba queda como ejercicio). Se pueden utilizar como "relojes" en las simulaciones porque se ejecutan en el tiempo que definen (es decir, T(n) se ejecuta en T(n) pasos). Este hecho de por sí explica por qué algunos autores definen la jerarquía temporal sólo en términos de las funciones tiempo-construibles: ¿qué sentido tiene definir una clase DTIME(T(n)) si no existe un mecanismo para determinar si una MT resuelve un problema en tiempo T(n)? Todas las funciones con las que se trabaja habitualmente en la complejidad temporal, como n^k, 2^n, n!, etc., son tiempo-construibles. También se cumple que, entre otras operaciones, $T_1 + T_2$, $T_1.T_2$ y 2^{T1} son tiempo construibles si T_1 y T_2 lo son (la

prueba queda como ejercicio). La referencia al "buen comportamiento" de las funciones tiempo-construibles tiene que ver con que el uso de funciones no tiempo-construibles puede producir efectos no deseados en la jerarquía temporal. Por ejemplo, que existan MT con cotas de tiempo siempre mejorables (Teorema de Aceleración o *Speed Up Theorem*). Otro ejemplo, enunciado en el Teorema del Hueco o *Gap Theorem*, es la posibilidad de la existencia en la jerarquía de amplias franjas vacías de problemas, a pesar de estar delimitadas por funciones temporales muy distantes entre sí.
El siguiente teorema formaliza, teniendo en cuenta las consideraciones anteriores, con cuánto más tiempo se pueden resolver más problemas, y además que siempre se puede encontrar una clase temporal que incluya estrictamente a otra.

Teorema 6.1. Teorema de la jerarquía temporal

Demostramos a continuación dos resultados básicos de la jerarquía temporal, ámbos por diagonalización. En primer lugar probamos que

si $T_2(n) > T_1(n).\log T_1(n)$ para infinitos n, entonces $DTIME(T_1(n)) \neq DTIME(T_2(n))$

siendo $T_2(n)$ una función tiempo-construible. Luego planteamos cómo a partir de este resultado, se puede encontrar una clase temporal que incluya estrictamente a $DTIME(T_1(n)))$.
Para la prueba, construimos una MT M que trabaja en tiempo $T_2(n)$ y se comporta distinto que todas las MT que trabajan en tiempo $T_1(n)$, usando que $T_2(n)$ es tiempo-construible y significativamente mayor que $T_1(n).\log_2 T_1(n)$ cuando n tiende a infinito. De esta manera reconoce un lenguaje de $DTIME(T_2(n)) - DTIME(T_1(n))$. Dada un entrada w, con $|w| = n$, M trabaja de la siguiente forma:

1. Si w no es el código de una MT, rechaza.
2. Suponiendo que w es el código de alguna MT M_w, es decir $w = <M_w>$, M simula M_w a partir de w. Que M_w tenga más cintas que M no es problema, porque ya sabemos que con dos cintas alcanza para cualquier simulación (al costo de un factor $\log T_1(n)$ si M_w trabaja en tiempo $T_1(n)$). Tampoco es problema la diferencia de los tamaños de los alfabetos: se puede fijar en M una cantidad de símbolos para representar los de M_w (en este caso el costo es una constante). En

definitiva, M puede simular $T_1(n)$ pasos de M_w con $c.T_1(n).\log T_1(n)$ pasos, siendo c una constante que depende de M_w.

3. Para no excederse de los $T_2(n)$ pasos, M simula en simultáneo una MT que trabaja en tiempo exactamente $T_2(n)$, lo que es posible porque $T_2(n)$ es tiempo-construible.
4. M acepta w si y sólo si la simulación de M_w a partir de w se completa y M_w rechaza w.

Permitiendo, sin perder generalidad, que los códigos $w = <M_w>$ estén precedidos por cualquier cantidad de ceros (es decir, permitiendo que $|Q_w|$ tenga cualquier cantidad de ceros a izquierda, de acuerdo a cómo definimos la codificación de una MT en la Clase 3), entonces por la hipótesis, toda MT M_w que trabaja en tiempo $T_1(n)$ se puede codificar con una cadena w lo suficientemente grande que cumple

$$c.T_1(|w|).\log T_1(|w|) \leq T_2(|w|)$$

Para este caso de w, entonces M puede simular completamente la MT M_w asociada, y así, por la construcción descripta, se cumple que w está en L(M) si y sólo si w no está en $L(M_w)$. De esta manera, $L(M) \neq L(M_w)$ para toda MT M_w que trabaja en tiempo $T_1(n)$, es decir que $L(M) \in DTIME(T_2(n)) - DTIME(T_1(n))$.

En particular, si $T_1(n) = O(T_2(n))$, entonces $DTIME(T_1(n)) \subset DTIME(T_2(n))$. Por ejemplo, se cumple $DTIME(n^k) \subset DTIME(n^{k+1})$ para todo número natural k. En cambio, si $T_1(n) \neq O(T_2(n))$, para encontrar una clase que incluya estrictamente a $DTIME(T_1(n))$ se puede definir una función $T_3(n) = \max(T_1(n), T_2(n))$ para todo n, es decir que T_3 se quede con el máximo de T_1 y T_2 para todo n; de esta manera se obtiene $DTIME(T_1(n)) \subset DTIME(T_3(n))$.

El segundo resultado, que probamos a continuación, refuerza el concepto de que la jerarquía temporal es *densa*, es decir que siempre se puede definir una clase de problemas mayor. En este caso no es necesario recurrir a las funciones tiempo-construibles. Demostramos que si T(n) es una función total computable, entonces existe un lenguaje recursivo L (o de EXP según nuestra convención) tal que $L \notin DTIME(T(n))$. Se trata del lenguaje $L = \{w_i \mid w_i$ no es aceptado por la MT M_i en $T(|w_i|)$ pasos$\}$, considerando como siempre el orden canónico:

- Se cumple que $L \in R$. La siguiente MT M reconoce L y se detiene siempre: dada una entrada v, M calcula i tal que $v = w_i$, genera $<M_i>$, calcula $k = T(|w_i|)$, y simula k pasos de M_i a partir de w_i, aceptando si y sólo si M_i no acepta w_i. Claramente L(M) = L, y M se detiene siempre (la prueba queda como ejercicio).
- Y se cumple que $L \notin DTIME(T(n))$. Supongamos que $L \in DTIME(T(n))$. Sea M_i una MTD que reconoce L en tiempo T(n):
  a. Si $w_i \in L$, entonces M_i acepta w_i en tiempo $T(|w_i|)$, pero por la definición de L se cumple que $w_i \notin L$ (absurdo).
  b. Si $w_i \notin L$, entonces M_i no acepta w_i, en particular no lo acepta en tiempo $T(|w_i|)$, pero por la definición de L se cumple que $w_i \in L$ (absurdo).

Fin de Teorema

Como corolario de otro resultado en el marco de la jerarquía temporal, conocido como el Teorema de la Unión (no lo vamos a considerar), existe una función total computable T(n) tal que DTIME(T(n)) = P. De esta manera, por el segundo resultado del teorema anterior se deduce que la clase P está incluida estrictamente en la clase EXP. De igual modo se prueba la inclusión estricta de NP en EXP.

Ejercicios de la Clase 6

1. Sea la siguiente definición alternativa a la utilizada en la Clase 6: $[f(n) = O(g(n))] \leftrightarrow [\exists c > 0$ 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.
2. Mostrar que en el marco de la complejidad temporal, las RAM constituyen un modelo de ejecución razonable, y la matriz de adyacencia es una representación razonable de un grafo.
3. Probar que si $T_1(n) = O(T_2(n))$, entonces $DTIME(T_1(n)) \subseteq DTIME(T_2(n))$.
4. Probar que las funciones tiempo-construibles son totales y computables. Comentar por qué la recíproca podría no cumplirse.
5. Sea M una MTN que si acepta una cadena w, al menos en una de sus computaciones la acepta en a lo sumo T(|w|) pasos, siendo T una función tiempo-construible. Probar que $L(M) \in NTIME(T(n))$.
6. Probar que las funciones n^2, 2^n y n! son tiempo construibles, y que si T_1 y T_2 son tiempo-construibles también los son $T_1 + T_2$, $T_1.T_2$, y 2^{T1}.

7. Completar la prueba del Teorema 6.1.
8. Considerando que las funciones polinomiales son tiempo-construibles, probar:
   i. $DTIME(2^n) \subset DTIME(n^2.2^n)$.
   ii. Para todo $k \geq 0$, $DTIME(n^k) \subset DTIME(n^{k+1})$.
9. Construir una MT M que genere los códigos de todas las MT M_i tales que a partir de $<M_i>$ trabajan en tiempo exactamente $T(|<M_i>|)$, siendo T una función tiempo-construible.
10. Probar (mediante una reducción desde HP) que no es decidible el problema de determinar si un lenguaje L pertenece a una clase DTIME(T(n)), dados cualquier lenguaje L y cualquier función tiempo-construible T.

Clase 7. Las clases P y NP

En esta clase presentamos ejemplos representativos de problemas de P y NP, y algunas características de estas clases.

Ya hemos descripto lenguajes y funciones que se reconocen o calculan en tiempo determinístico polinomial: el lenguaje de las cadenas a^ib^i con $i \geq 1$, la resta de dos números naturales, el lenguaje de los palíndromes, etc. En cada caso se probó por construcción la existencia de una MTD que trabaja en tiempo $O(n^k)$, siendo n la longitud de las entradas y k una constante.

Se presentan a continuación tres problemas clásicos de resolución determinística polinomial, de la teoría de grafos, la aritmética y la lógica, no tan simples como los mencionados antes. Para distinguirla de P, llamaremos FP a la clase de las funciones que se calculan en tiempo determinístico polinomial.

Ejemplo 7.1. El problema del camino mínimo en un grafo está en P

El problema (de decisión) del camino mínimo en un grafo consiste en determinar si en un grafo existe un camino entre dos vértices v_1 y v_2, de longitud a lo sumo K.

Un grafo se representará por un par (V, E) como se describió previamente, utilizando números en binario para la identificación de los vértices, y el símbolo # como separador.

La idea general del algoritmo propuesto se basa en lo siguiente. Si $A^h(i, j)$ es la longitud del camino mínimo entre los vértices i y j que no pasa por ningún vértice mayor que h, entonces se cumple

$$A^{h+1}(i, j) = \min(A^h(i, h+1) + A^h(h+1, j), A^h(i, j))$$

Naturalmente, el camino mínimo entre i y j que no pasa por ningún vértice mayor que h+1, pasa o no pasa por el vértice h+1. La siguiente MTD M, basada en la igualdad anterior, trabaja en tiempo polinomial y reconoce el lenguaje SP (por *shortest path* o camino mínimo) que representa el problema, siendo SP = {(G, v_1, v_2, K) | G es un grafo y tiene un camino entre sus vértices v_1 y v_2 de longitud a lo sumo K}. Dada una entrada w = (G, v_1, v_2, K), M obtiene $A^m(v_1, v_2)$, el camino mínimo en G entre v_1 y v_2, y acepta si y sólo si $A^m(v_1, v_2) \leq K$. Se utilizan matrices A^i de m x m para almacenar los valores que se van calculando:

1. Si w no es una entrada válida, rechaza.
2. Para todo i, j ≤ m, hace:

   Si G incluye un arco (i, j), entonces $A^1[i, j] := 1$, si no $A^1[i, j] := m$.
3. Para todo h = 2, 3, …, m, hace:

   Para todo i, j ≤ m, hace:

   $A^h[i, j] := \min(A^{h-1}[i, h] + A^{h-1}[h, j], A^{h-1}[i, j])$.
4. Si $A^m[v_1, v_2] \leq K$, entonces acepta, si no rechaza.

En el paso 1 hay que verificar fundamentalmente que G es válido (los vértices de V son 1, …, m, los arcos de E no se repiten y sus vértices están en V), lo que lleva tiempo $O(|V|) + O(|E|^2) + O(|V||E|)$, y que v_1 y v_2 son vértices distintos válidos y K un número natural menor que m, lo que lleva tiempo lineal. Así, el tiempo consumido por este paso es $O(n^2)$. La asignación $A^1[i, j] := m$ en el paso 2 significa que $A^1[i, j]$ recibe un valor muy grande, seguro que mayor que la longitud del camino mínimo. Claramente, el tiempo consumido por los pasos 2 a 4 es $O(m^3) = O(n^3)$. Por lo tanto, la MTD M hace $O(n^2) + O(n^3) = O(n^3)$ pasos. Queda como ejercicio probar que efectivamente L(M) = SP.

Fin de Ejemplo

El siguiente es un ejemplo que considera una función de la clase FP. Corresponde al cálculo del máximo común divisor de dos números naturales. Es uno de los algoritmos no triviales más antiguos, atribuido a Euclides.

Ejemplo 7.2. El problema del máximo común divisor está en FP

El máximo común divisor de dos números naturales *a* y b, denotado con mcd(a, b), es el máximo número natural que divide a los dos. Por ejemplo, mcd(30, 24) = 6. Se cumple que si r es el resto de la división de *a* sobre b, es decir si r = a mod b, con a ≥ b, entonces

$$mcd(a, b) = mcd(b, r)$$

En base a esta idea se presenta la siguiente MTD M, que trabaja en tiempo polinomial. M calcula en la variable x el valor mcd(a, b), con a ≥ b:

1. Si b = 0, entonces hace x := a, y acepta.
2. Hace r := a mod b.
3. Hace a := b.
4. Hace b := r.
5. Vuelve al paso 1.

Veamos que la MT M itera a lo sumo log b veces. Sean (a_{k-1}, b_{k-1}), (a_k, b_k), (a_{k+1}, b_{k+1}), tres pares sucesivos calculados por M:

- Se cumple $a_k = q.b_k + b_{k+1}$, para algún $q \geq 1$. Por lo tanto, $a_k \geq b_k + b_{k+1}$.
- Como $b_{k-1} = a_k$, entonces $b_{k-1} \geq b_k + b_{k+1}$.
- A partir de lo anterior se puede probar que $b = b_0 \geq 2^{k/2} b_k$, para todo número natural par $k \geq 2$. Esto significa que k, que representa el número de pasos de la MT M, está acotado por $\log_2 b$.

Así, M trabaja en tiempo determinístico lineal con respecto a la longitud de sus entradas. Queda como ejercicio probar que efectivamente M calcula en x el valor mcd(a, b).

Fin de Ejemplo

Con el tercer y último ejemplo de problema en P, volvemos a la satisfactibilidad en la lógica. Vamos a considerar distintas variantes de dicho problema a lo largo de esta parte del libro.

Ejemplo 7.3. El problema 2-SAT está en P

El problema 2-SAT consiste en determinar si una fórmula booleana φ (con una sintaxis determinada que enseguida especificamos) es satisfactible, es decir, si existe una asignación de valores de verdad para φ que la hace verdadera. Una fórmula booleana se define inductivamente de la siguiente manera:

1. Una variable x es una fórmula booleana.

2. Si φ_1 y φ_2 son fórmulas booleanas, también lo son $(\varphi_1 \vee \varphi_2)$, $(\varphi_1 \wedge \varphi_2)$ y $\neg\varphi_1$. Los paréntesis redundantes pueden omitirse.

Si una fórmula boolana φ es una conjunción de disyunciones de *literales*, siendo un literal una variable o una variable negada, se dice que φ tiene o está en la *forma normal conjuntiva*. Es el caso, por ejemplo, de

$$\varphi = (x_1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_4 \vee x_2) \wedge (x_1 \vee x_7 \vee x_5)$$

Las disyunciones se denominan *cláusulas*. En particular, 2-SAT considera sólo fórmulas booleanas en la forma normal conjuntiva con dos literales por cláusula. El lenguaje que representa el problema es 2-SAT = {φ | φ es una fórmula booleana satisfactible, en la forma normal conjuntiva, con dos literales por cláusula}. Para probar que 2-SAT está en P, vamos a construir una MTD M que lo reconoce en tiempo polinomial. La idea general es la siguiente. M empieza asignando arbitrariamente el valor verdadero a alguna variable x, completa consistentemente todas las asignaciones que puede, barre una a una las cláusulas $c = (\varphi_1 \vee \varphi_2)$ con al menos un literal asignado, y las procesa adecuadamente (por ejemplo, si uno de los dos literales tiene el valor verdadero, declara satisfecha a la cláusula). Si no rechaza por detectar la instatisfactibilidad de la fórmula completa, M repite el proceso a partir de otra asignación arbitraria a alguna variable x, hasta decidir que la fórmula está o no en 2-SAT. Formalmente, dada una entrada φ, el conjunto C de cláusulas (al comienzo todas declaradas insatisfechas), y el conjunto V de variables (al comienzo todas declaradas no asignadas), la MT M hace:

1. Si φ no es una entrada válida sintácticamente, rechaza.
2. Si el conjunto V está vacío, acepta.
3. Dada una variable x de V, hace x := verdadero.
   Hace primer-valor := verdadero.
   Mientras C tenga una cláusula $c = (\varphi_1 \vee \varphi_2)$ insatisfecha con al menos un literal asignado, hace:
      Si φ_1 = verdadero, o bien φ_2 = verdadero, entonces declara a la cláusula c satisfecha.
      Si no, si φ_1 = falso, y también φ_2 = falso, entonces:
         Si primer-valor ≠ verdadero, rechaza.

Si no, declara insatisfechas a todas las cláusulas de C
declara no asignadas a todas las variables de V
hace x := falso
hace primer-valor := falso.
Si no, si φ_1 = falso, entonces hace φ_2 := verdadero
Si no, entonces hace φ_1 := verdadero.
Elimina de C las cláusulas satisfechas, y de V las variables asignadas.

4. Vuelve al paso 2.

El análisis sintáctico de la fórmula φ en el paso 1 es lineal, y el tiempo consumido por los pasos 2 a 4 es $O(|V||C|) = O(n^2)$. Por lo tanto, la MTD M trabaja en tiempo $O(n^2)$. Queda como ejercicio probar que efectivamente L(M) = 2-SAT.

Fin de Ejemplo

La clase P es cerrada con respecto a la unión, intersección y complemento, entre otras operaciones entre lenguajes (la prueba queda como ejercicio). Además de poblarla mediante la construcción de MTD que trabajan en tiempo polinomial, otra manera es utilizando reducciones de problemas, pero ahora acotadas temporalmente, como veremos en la próxima clase.

Para comenzar con el análisis de la clase NP, presentamos a continuación un par de ejemplos, que corresponden a problemas clásicos sobre grafos. Se prueba fácilmente que pertenecen a NP, y no se les conoce resolución determinística polinomial.

Ejemplo 7.4. El problema del circuito de Hamilton está en NP

Este problema consiste en 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 $i_1, \ldots, i_m$ de V, tal que los arcos $(i_1, i_2), \ldots, (i_{m-1}, i_m), (i_m, i_1)$ pertenecen a E. Es decir, un circuito de Hamilton en un grafo recorre todos sus vértices sin repetirlos, arrancando y terminando en un mismo vértice.

Sea HC (por *Hamiltonian circuit* o circuito hamiltoniano) el lenguaje que representa el problema, con HC = {G | G es un grafo que tiene un circuito de Hamilton}. El algoritmo determinístico natural para reconocer HC chequea en el peor caso todas las permutaciones de V, para detectar si una de ellas es un circuito de Hamilton. Tarda

tiempo exponencial: hay m! permutaciones de V, por lo que el tiempo de ejecución es al menos

$$O(m!) = O(m^n)$$

Para probar que HC está en NP, construimos una MTN M que reconoce HC en tiempo polinomial. Dada una entrada G = (V, E), M hace:

1. Si G es un grafo inválido, rechaza.
2. Genera no determinísticamente una cadena C, con símbolos de {0, 1, #}, de tamaño |V|.
3. Acepta si y sólo si C es un circuito de Hamilton de G.

Se cumple HC = L(M).

$G \in HC \leftrightarrow$ G tiene un circuito de Hamilton $\leftrightarrow$ M genera en el paso 2 de alguna computación un circuito de Hamilton $\leftrightarrow$ M acepta $G \leftrightarrow G \in L(M)$.

M trabaja en tiempo no determinístico polinomial.

- Ya se indicó que el tiempo del paso 1 es $O(n^2)$.
- En el paso 2, la generación de un cadena C de tamaño |V| tarda $O(|V|) = O(n)$.
- En el paso 3 hay que chequear que $C = i_1, \ldots, i_m$ es una permutación de V, y que los arcos $(i_1, i_2), \ldots, (i_{m-1}, i_m), (i_m, i_1)$ están en E. Esto tarda $O(|V|^2) + O(|V||E|) = O(n^2)$.

Por lo tanto, M trabaja en tiempo $O(n^2)$.

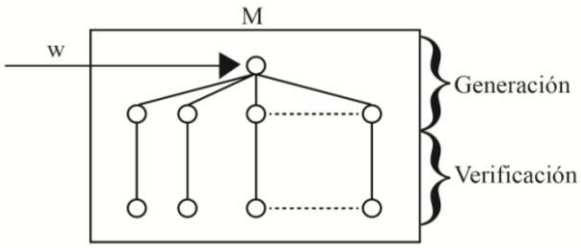
Fin de Ejemplo

La forma de una MTN M que trabaja en tiempo polinomial, construida para demostrar que L(M) pertenece a la clase NP, siendo L(M) la representación de un problema (de decisión), es siempre la misma. Dada una entrada w, M hace:

1. Genera no determinísticamente, en tiempo polinomial, una posible solución del problema (solución en el sentido más amplio: por ejemplo, en el caso del problema de primalidad, la cadena generada puede simplemente establecer que w es un número primo).

2. Chequea determinísticamente en cada computación, en tiempo polinomial, si lo generado en el paso 1 es efectivamente una solución del problema. El chequeo incluye la validación sintáctica de la entrada w, que se puede ejecutar al principio para optimizar el algoritmo.

Enseguida vamos a presentar una definición equivalente de la clase NP, relacionada con esta última consideración. La figura siguiente ilustra la forma de las MTN referida recién:



Ejemplo 7.5. El problema del clique está en NP

Este problema consiste en determinar si un grafo tiene un clique de tamaño (al menos) K. Un clique de tamaño K en un grafo G es un subgrafo completo de G con K vértices. También se lo puede definir como un conjunto de vértices $C = \{i_1, \ldots, i_K\} \subseteq V$, tal que para todo par de vértices v, v' de C se cumple que $(v, v') \in E$.

El lenguaje que representa el problema es CLIQUE = {(G, K) | G es un grafo que tiene un clique de tamaño K}. El algoritmo determinístico natural para reconocer CLIQUE consiste en recorrer, en el peor caso, todos 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 ejecución es al menos

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

La siguiente MTN M reconoce CLIQUE en tiempo polinomial, lo que prueba que CLIQUE está en NP. Dada una entrada w = (G, K), M hace:

1. Si w es inválida, rechaza.

2. Genera no determinísticamente una cadena C, con símbolos de {0, 1, #}, de tamaño a lo sumo |V|.
3. Acepta si y sólo si C es un clique en G de tamaño K.

Queda como ejercicio probar que CLIQUE = L(M), y que M trabaja en tiempo no determinístico polinomial.

Fin de Ejemplo

Notar en el último ejemplo que si K no formara parte de las entradas de la MT M, es decir, si fuera una constante, entonces el problema estaría en P: hay m! / ((m – K)!.K!) = $O(n^K)$ subconjuntos de K vértices en V, y el chequeo de que los mismos constituyen cliques se puede hacer en tiempo determinístico polinomial (podría discutirse de todos modos si para un K muy grande, por ejemplo 1000, el tiempo de la MTD construida puede considerarse aceptable).

Por otro lado, el complemento del lenguaje CLIQUE no parece estar ni siquiera en NP. Sea NOCLIQUE = {(G, K) | G es un grafo que no tiene un clique de tamaño K}. Aún con la posibilidad del no determinismo, deberían recorrerse todos los subconjuntos de K vértices para aceptar o rechazar adecuadamente. No se conoce algoritmo no determinístico polinomial para reconocer NOCLIQUE. De hecho se conjetura que la clase NP no es cerrada con respecto al complemento. Si CO-NP es la clase de los lenguajes complemento de los de NP, sólo vamos a establecer por ahora que se cumple

$$P \subseteq NP \cap CO\text{-}NP$$

La prueba de esta inclusión queda como ejercicio. La relación entre las clases NP y CO-NP se trata en la Clase 9.

Los algoritmos asociados a los problemas de NP de los que no se conocen resoluciones eficientes se basan, como en los dos ejemplos precedentes, en explorar el espacio completo de posibles soluciones, o al menos una parte amplia del mismo (en contraste con lo que vimos en los algoritmos desarrollados al comienzo de la clase, que resuelven problemas de P), lo que los lleva a consumir tiempo exponencial. Esta es una constante observada en cientos de problemas de NP, en áreas tan diversas como la teoría de grafos, el álgebra, la aritmética, la teoría de autómatas, la lógica, etc. Si se probara P = NP se estaría demostrando entonces que estos algoritmos de "fuerza bruta" pueden

sustituirse por sofisticadas resoluciones ejecutadas eficientemente. A propósito, como se indica más adelante, la técnica de simulación de MT no sirve para probar P = NP (tampoco sirve la diagonalización para demostrar lo contrario).

Una definición alternativa de la clase NP, esta vez sin utilizar MTN, contribuye sobremanera a entender el tipo de problemas que la integran. Según dicha definición, un lenguaje L que pertenece a NP goza de la propiedad de que cada una de sus cadenas w tiene al menos un *certificado suscinto* z que atestigua su pertenencia a L. Se define que z es un certificado suscinto de w si:

- Es una cadena de tamaño polinomial con respecto al tamaño de w.
- El predicado R(w, z), que expresa que z es un certificado suscinto de w, se decide en tiempo determinístico polinomial.

Dicho de otra manera: para toda instancia positiva de un problema de NP existe al menos un certificado suscinto de que es positiva. Podemos no saber cómo encontrar el certificado eficientemente, pero seguro que existe. Por ejemplo, en el caso del problema del circuito de Hamilton, los grafos con circuitos de Hamilton tienen certificados suscintos: son los mismos circuitos de Hamilton. Otro ejemplo lo constituye el problema de primalidad, recién mencionado: todo número primo tiene un certificado suscinto de su primalidad.

Esta visión alternativa de la clase NP justifica su riqueza. Muchísimos problemas comparten la propiedad de que sus posibles soluciones son a lo sumo polinomialmente más grandes que sus instancias, y de que se puede decidir eficientemente si una posible solución es efectivamente una solución. Asimismo, el concepto refuerza la conjetura $P \neq NP$: intuitivamente es más fácil probar que una posible solución es efectivamente una solución (lo que se requiere para demostrar la pertenencia a NP), en comparación con construir una solución (lo que se requiere para demostrar la pertenencia a P).

Se demuestra a continuación que las dos definiciones de NP son equivalentes.

Teorema 7.1. Definición alternativa de la clase NP

Vamos a probar que un lenguaje L pertenece a la clase NP, si y sólo si L se puede definir de la siguiente manera:

$$L = \{w \mid \exists z: |z| \leq p(|w|) \wedge R(w, z)\}$$

siendo p un polinomio y R un predicado de dos argumentos que se decide en tiempo determinístico polinomial (diremos también en este caso que $R \in P$). Es trivial, y queda como ejercicio, que si existe un predicado de dos argumentos R en P, un polinomio p, y un lenguaje L como el definido, entonces L está en NP.

Probamos a continuación el sentido contrario. 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 tal que su relación de transición tiene grado 2 (la demostración queda como ejercicio). Vamos a construir una MTN M_3 equivalente a M_2, que trabaja también en tiempo polinomial, y de cuya forma se inferirá la existencia de los componentes que estamos buscando. Suponiendo que el tiempo de ejecución de M_2 es el polinomio p(n), y que su relación de transición es Δ_2, M_3 hace, a partir de una entrada w:

1. Calcula p(|w|) y genera no determinísticamente una cadena z de {0, 1}* de longitud p(|w|).
2. Decide determinísticamente un predicado R(w, z), lo cual consiste en simular M_2 a partir de w, de acuerdo a la secuencia z de ceros y unos generada (por ejemplo, si z empieza con 0110, M_3 simula el primer paso de M_2 tomando la primera alternativa de Δ_2, luego simula el segundo paso de M_2 tomando la segunda alternativa de Δ_2, etc).

Claramente, $L(M_2) = L(M_3)$ (la prueba queda como ejercicio). Además, la MTN M_3 trabaja en tiempo polinomial: el paso 1 tarda tiempo no determinístico O(p(|w|)), y el paso 2 tarda tiempo determinístico O(p(|w|)). De este modo, $L = L(M_3)$ se puede expresar de la manera requerida.

Fin de Teorema

Dos últimas caracterizaciones que presentamos a continuación sirven para comparar P y NP. La primera de ellas se refiere al impacto que produce especializar y generalizar problemas de NP. La especialización de un problema de NP puede tener el efecto de simplificarlo. Por ejemplo, en el caso del problema k-SAT, de satisfactibilidad de fórmulas booleanas en la forma normal conjuntiva con k literales por cláusula, cuando k = 2 el problema está en P, como ya demostramos en el Ejemplo 7.3. Por otro lado, en la

clase siguiente vamos a demostrar que cuando k = 3, el problema pertenece a la subclase de los problemas más difíciles de NP, conocida como NPC o la clase de los problemas *NP-completos*. Justamente, con un efecto contrario al de la especialización, generalizar un problema puede complicarlo. En síntesis, en el caso del problema k-SAT, la frontera entre lo eficiente e ineficiente (naturalmente asumiendo P ≠ NP) está entre los parámetros k = 2 y k = 3.

Otro ejemplo es el problema de la k-coloración, que consiste en determinar si se pueden colorear con k colores los vértices de un grafo, de manera tal que dos vértices vecinos no tengan el mismo color. Como en el caso anterior, se prueba que para k = 2 el problema está en P (lo vamos a probar en la próxima clase mediante una reducción de problemas), y que para k = 3 es NP-completo.

Aún cuando no aplican las nociones de especialización y generalización, existen numerosos casos de problemas con definiciones muy parecidas, que sin embargo difieren drásticamente en su complejidad temporal. Por ejemplo, el problema de la programación lineal consiste en determinar si un sistema de inecuaciones lineales con coeficientes enteros tiene solución, y se resuelve eficientemente, mientras que si se exige que la solución sea estrictamente entera (en este caso se lo conoce como programación lineal entera), pasa a ser NP-completo. Esta diferencia es razonable, porque numerosos problemas NP-completos se expresan fácilmente mediante inecuaciones lineales sobre los números enteros.

Otro caso lo constituyen el cálculo del determinante y el cálculo de la permanente de una matriz. Si bien se definen de una manera muy similar, el segundo es mucho más difícil que el primero. La diferencia se puede justificar por lo siguiente. El problema de correspondencia perfecta (o *matching* perfecto) en un grafo bipartito G (se lo llama así porque los vértices se particionan en dos conjuntos, de modo tal que los arcos sólo van de un conjunto a otro) consiste en determinar si G tiene un subconjunto de arcos no adyacentes que cubre todos sus vértices. Este problema está en P, y se puede resolver mediante un determinante. Por su parte, calcular la cantidad de correspondencias perfectas en grafos bipartitos, que es mucho más difícil, se puede resolver por medio de una permanente.

La última caracterización de P y NP que queremos destacar en esta clase se relaciona con la expresividad de la lógica (esta aproximación se conoce como *complejidad descriptiva*). Se demuestra que NP coincide con los problemas que pueden especificarse con la *lógica existencial de segundo orden* (que enseguida describiremos). De esta

manera, encontrando otra lógica para P se probaría P ≠ NP. Si bien no han habido avances en este sentido, nos parece ilustrativo extendernos un poco sobre esta cuestión, en que se relacionan íntimamente los problemas sobre grafos, la lógica, y las clases P y NP.

Ejemplos de fórmulas de primer orden para expresar propiedades de grafos finitos G son $\forall xG(x,x)$, $\forall x\forall y(G(x,y) \rightarrow G(y,x))$, y $\exists x\forall yG(y,x)$, que expresan respectivamente que G es reflexivo, simétrico, y que tiene un vértice al que llegan arcos desde el resto. Las fórmulas no utilizan símbolos de función, y además de la igualdad cuentan sólo con el símbolo de predicado binario G. Se prueba fácilmente que las fórmulas φ de este tipo se pueden chequear en tiempo determinístico polinomial:

- Si φ es atómica, es decir si tiene la forma G(x,x) o G(x,y), claramente se puede chequear eficientemente.
- Si $\varphi = \neg\varphi_1$, por hipótesis inductiva φ_1 se puede chequear eficientemente, y por lo tanto también su negación.
- Si $\varphi = \varphi_1 \vee \varphi_2$, por hipótesis inductiva φ_1 y φ_2 se pueden chequear eficientemente, y por lo tanto también su disyunción.
- La prueba para $\varphi = \varphi_1 \wedge \varphi_2$ es similar a la anterior.
- Finalmente, si $\varphi = \forall x\varphi_1$, por hipótesis inductiva φ_1 se puede chequear eficientemente, y por lo tanto también $\forall x\varphi_1$, iterando el mismo algoritmo sobre cada uno de los vértices.

No todos los problemas sobre grafos pertenecientes a la clase P se pueden expresar con fórmulas de primer orden. Un ejemplo es el problema de la alcanzabilidad (determinar si existe un camino en un grafo de un vértice x a un vértice y). En este caso se debe recurrir a una lógica de mayor expresividad, la lógica de segundo orden, que permite agregar variables para funciones y predicados, y cuantificadores que operan sobre dichas variables. Particularmente, las fórmulas de la lógica existencial de segundo orden tienen la forma $\exists P\varphi$, siendo φ una subfórmula de primer orden. Por ejemplo, con la fórmula $\exists P\forall x\forall y(G(x,y) \rightarrow P(x,y))$ se puede expresar que G es un subgrafo del grafo P. Por su parte, la fórmula

$$\exists P\forall u\forall v\forall w((G(u,v) \rightarrow P(u,v)) \wedge P(u,u) \wedge ((P(u,v) \wedge P(v,w)) \rightarrow P(u,w)) \wedge \neg P(x, y))$$

establece que G es un subgrafo de P, P es reflexivo, transitivo, y no tiene un arco del vértice x al vértice y, y por lo tanto, que G no tiene un camino entre los vértices x e y, que es la propiedad inversa de la alcanzabilidad. Con algunas manipulaciones más se llega a expresar la alcanzabilidad. Este problema está en P, pero con la lógica existencial de segundo orden se pueden expresar también problemas mucho más difíciles, como el de la 3-coloración, que es NP-completo: utilizando tres símbolos de predicado unarios, R por rojo, A por amarillo y V por verde (para hacer más clara la expresión, en lugar de usar un solo símbolo P con más aridad), una fórmula apropiada que lo especifica es

$$\exists R \exists A \exists V \forall x((R(x) \vee A(x) \vee V(x)) \wedge \forall y(G(x,y) \rightarrow (\neg(R(x) \wedge R(y)) \wedge \neg(A(x) \wedge A(y)) \wedge \neg(V(x) \wedge V(y)))))$$

es decir que todo vértice x es rojo, amarillo o verde, y ningún vecino de x puede tener su mismo color. En realidad, como lo enuncia el Teorema de Fagin, todos los lenguajes de NP y sólo ellos se reducen a propiedades sobre grafos expresables con fórmulas existenciales de segundo orden. Ya vimos que, en cambio, no se puede relacionar de esta manera a P con la lógica de primer orden. Otro lenguaje candidato en este último sentido es el de las *fórmulas de Horn* de la lógica existencial de segundo orden, en que el fragmento de primer orden φ de $\exists P\varphi$ cumple que:

- Está en la forma prenex (todos los cuantificadores están al comienzo).
- Sus cuantificadores son sólo universales.
- La matriz, es decir el fragmento que le sigue a los cuantificadores, es una conjunción de disyunciones cada una de las cuales tiene a lo sumo una fórmula atómica no negada que incluye al símbolo P (*cláusulas de Horn*).

Por ejemplo, la fórmula del complemento de la alcanzabilidad que presentamos antes tiene dicha sintaxis, con cláusulas de Horn $\neg G(u,v) \vee P(u,v)$, $P(u,u)$, $\neg P(u,v) \vee \neg P(v,w) \vee P(u,w)$, y $\neg P(x, y)$. Esta lógica tampoco cubre P. En este caso se debe a que en φ no puede definirse la condición de ser sucesor: además del predicado G, se necesita otro predicado, digamos S, para representar el sucesor.

Ejercicios de la Clase 7

1. Completar la prueba del Ejemplo 7.1.
2. Completar la prueba del Ejemplo 7.2.
3. Completar la prueba del Ejemplo 7.3.
4. Probar que la clase P es cerrada con respecto a la unión, la intersección y el complemento.
5. Probar que $P \subseteq NP \cap CO\text{-}NP$.
6. Sean $f \in FP$ y $L \in P$. Probar que $f^{-1}(L) = \{x \mid f(x) \text{ está en } L\} \in P$.
7. Determinar si los siguientes lenguajes pertenecen a P:
   i. Las fórmulas booleanas en la forma normal conjuntiva satisfactibles con asignaciones de a lo sumo diez variables verdaderas.
   ii. Las fórmulas booleanas satisfactibles en la *forma normal disyuntiva*, es decir las fórmulas satisfactibles que son disyunciones de conjunciones de literales.
8. Probar si existen MT que trabajan en tiempo determinístico polinomial para:
   i. Determinar si una fórmula booleana es una *tautología*, es decir si es satisfactible por toda asignación de valores de verdad.
   ii. Determinar si una *fórmula booleana de Horn* es satisfactible, donde las fórmulas booleanas de Horn son fórmulas booleanas en la forma normal conjuntiva tal que cada cláusula tiene a lo sumo una variable no negada.
   iii. Transformar una fórmula booleana φ_1 en una fórmula booleana φ_2 en la forma normal conjuntiva que es satisfactible si y sólo si φ_1 lo es.
   iv. Determinar si es satisfactible una fórmula booleana en la forma normal conjuntiva en que cada variable aparece hasta dos veces.
   v. Resolver el problema 3-SAT restringido a que cada variable aparece hasta tres veces.
   vi. Transformar una fórmula booleana cuantificada en una fórmula equivalente en la forma prenex, tal que la matriz es una expresión booleana en la forma normal conjuntiva con tres literales por cláusula.
9. Completar la prueba del Ejemplo 7.5.
10. Probar que la clase NP es cerrada con respecto a la unión y la intersección.
11. Completar la prueba del Teorema 7.1.
12. Probar que los siguientes lenguajes pertenecen a la clase NP:

i. DOM = {(G, K) | el grafo G tiene un conjunto dominante de K nodos}. Un subconjunto de vértices C de un grafo G es un conjunto dominante de G, si todo otro vértice de G es adyacente a algún vértice de C.

ii. ISO = {(G_1, G_2) | G_1 y G_2 son grafos isomorfos}. Dos grafos son isomorfos si son iguales salvo por los nombres de sus arcos (pares de vértices).

13. Sea FNP el conjunto de las funciones computadas en tiempo polinomial por las MTN (se asume que todas las computaciones calculan el mismo valor). Probar que P = NP si y sólo si FP = FNP.
14. Mostrar cómo enumerar las clases P y NP.
15. Una función f es *honesta*, si para toda cadena z de su codominio existe una cadena x de su dominio tal que f(x) = z y $|x| \leq p(|y|)$, siendo p un polinomio. Se define además que dadas dos funciones f y g, g es una función *inversa derecha* de f si toda cadena z del codominio de f cumple f(g(z)) = z. Probar que si una función tiene una función inversa derecha computable en tiempo polinomial, entonces es honesta.
16. Sea f una función honesta computable en tiempo polinomial. Probar que existe una MTN M que trabaja en tiempo polinomial tal que L(M) coincide con el codominio de f.

Clase 8. Problemas NP-completos

Hemos anticipado en la clase anterior la existencia de una subclase de NP que contiene sus problemas más difíciles, la subclase NPC de los problemas NP-completos. Para precisar este concepto, que es el tema central de esta clase, introducimos primero las reducciones polinomiales de problemas. Las reducciones deben entenderse como m-reducciones (recordar que las renombramos así en el Tema 5.4 para diferenciarlas de las Turing-reducciones, sobre las que volveremos en la Clase 10), es decir que son las reducciones que tratamos en profundidad en la Clase 4.

Definición 8.1. Reducción polinomial de problemas

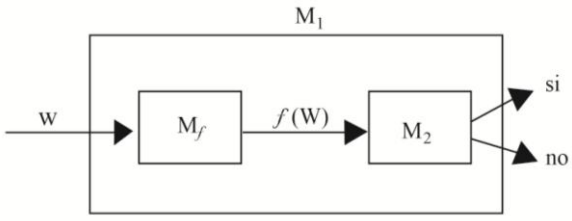
Una *reducción polinomial* del lenguaje L_1 al lenguaje L_2 es una reducción de L_1 a L_2 computable en tiempo determinístico polinomial. Se la denomina también *Karp-reducción*. Como siempre, M_f identifica la MT M que computa la función de reducción f (en este caso en tiempo determinístico polinomial, es decir que $f \in FP$).
Utilizaremos la notación $L_1 \alpha_P L_2$ para expresar que existe una reducción polinomial del lenguaje (o problema) L_1 al lenguaje (o problema) L_2.

Fin de Definición

Como en las reducciones generales (las que no tienen cota temporal), en que si se cumple $L_1 \alpha L_2$ entonces L_2 es tan o más difícil que L_1 desde el punto de vista de la computabilidad, con las reducciones polinomiales se puede establecer la misma relación, ahora desde el punto de vista de la complejidad temporal. Esto se formaliza en el siguiente teorema.

Teorema 8.1. Si L_2 está en P (NP) y $L_1 \alpha_P L_2$, entonces L_1 está en P (NP)

Demostramos sólo el caso de P, el de NP se prueba de manera similar y queda como ejercicio. Parte de la prueba ya se desarrolló en la demostración del teorema análogo para R (ver Teorema 4.1): para reconocer L_1, se construye una MTD M_1 componiendo una MTD M_f que computa la reducción de L_1 a L_2, con una MTD M_2 que reconoce L_2, es decir:



M_1 se detiene siempre porque M_f y M_2 se detienen siempre. Falta comprobar que M_1 trabaja en tiempo determinístico polinomial, y así que $L_1 \in P$:

- Dada una entrada w, M_f computa la reducción de L_1 a L_2 en a lo sumo $a.|w|^b$ pasos, con a y b constantes.
- Dada una entrada f(w), M_2 reconoce L_2 en a lo sumo $c.|f(w)|^d$ pasos, con c y d constantes.
- Así llegamos a que M_1 trabaja en tiempo determinístico polinomial: dada un entrada 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 $a.|w|^b$ pasos no puede generarse una cadena de más de $a.|w|^b$ símbolos), y luego completa su trabajo con a lo sumo $c.(a.|w|^b)^d$ pasos al simular M_2 a partir de f(w). Así, M_1 hace en total a lo sumo $a.|w|^b + c.(a.|w|^b)^d$ pasos, es decir $O(|w|^e)$ pasos, con e constante.

Fin de Teorema

Como corolario del teorema, dados dos lenguajes L_1 y L_2, si L_1 no está en P (NP), y existe una reducción polinomial de L_1 a L_2, entonces L_2 tampoco está en P (NP). Por lo tanto, podemos emplear también las reducciones polinomiales para probar que un problema no está en la clase P o en la clase NP. Al igual que α, la relación α_P es reflexiva y transitiva (la prueba queda como ejercicio). No es simétrica. Ya hemos probado que para todo lenguaje recursivo L se cumple L α L_U y no se cumple L_U α L. Para el caso α_P podemos considerar el mismo contraejemplo: la reducción considerada de L a L_U es lineal (a partir de una entrada w se genera una salida (<M>, w), siendo M una MT que reconoce L). El siguiente es un ejemplo de prueba de pertenencia a P por medio de una reducción polinomial.

Ejemplo 8.1. Reducción polinomial de 2-COLORACIÓN a 2-SAT

Los problemas de la 2-coloración y 2-SAT se presentaron en la clase pasada.

Sea 2-COLORACIÓN = {G | G es un grafo tal que sus vértices se pueden colorear con 2 colores de manera tal que dos vértices vecinos no tengan el mismo color} el lenguaje que representa el primer problema. El lenguaje 2-SAT, que representa el segundo problema, se describió en el Ejemplo 7.3, en el que también se probó que está en P. Se definió así: 2-SAT = {φ | φ es una fórmula booleana satisfactible, en la forma normal conjuntiva, con dos literales por cláusula}. Vamos a demostrar que también 2-COLORACIÓN está en P, reduciéndolo polinomialmente a 2-SAT.

Definición de la función de reducción.

A todo grafo válido G, la función de reducción f le asigna una fórmula booleana φ en la forma normal conjuntiva con dos literales por cláusula, de modo tal que por cada arco (i, k) de G construye dos cláusulas, de la forma $(x_i \vee x_k)$ y $(\neg x_i \vee \neg x_k)$.

La función f es total y pertenece a FP.

Claramente f es una función total. En particular, cuando un grafo es inválido, M_f genera una fórmula inválida sintácticamente (mantendremos el estándar de generar como salida inválida la cadena 1, que tarda O(1)). Además, f se computa en tiempo determinístico polinomial: la validación sintáctica inicial es cuadrática, y la generación de la fórmula booleana es lineal.

Se cumple $G \in$ 2-COLORACIÓN $\leftrightarrow \varphi \in$ 2-SAT.

Asociando dos colores c_1 y c_2 con los valores verdadero y falso, respectivamente, claramente los vértices de todo arco de G tienen colores distintos si y sólo si la conjunción de las dos cláusulas que se construyen a partir de ellos es satisfactible.

Fin de Ejemplo

El siguiente es otro ejemplo de reducción polinomial, esta vez entre dos problemas de NP de los que no se conocen resoluciones eficientes.

Ejemplo 8.2. Reducción polinomial de HC a TSP

El lenguaje HC, que representa el problema del circuito de Hamilton, se definió en la clase pasada de la siguiente manera: HC = {G | G es un grafo que tiene un circuito de Hamilton}.

Por su parte, se define TSP = {(G, B) | G es un grafo completo, sus 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* o problema del viajante de comercio) representa el problema del viajante de comercio, quien 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 de los arcos (en este caso longitudes), después de la representación de cada arco agregamos un separador # y un número natural en binario.

A continuación probamos que HC α_P TSP.

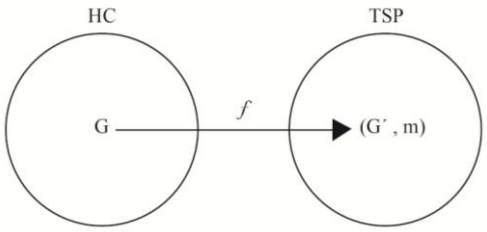
Definición de la función de reducción.

Para grafos válidos G se define

$$f(G) = (G', m)$$

donde 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 es la cantidad de vértices de G' (y de G).

La figura siguiente ilustra la reducción planteada:



La función f es total y pertenece a FP.

A partir de grafos inválidos M_f genera pares inválidos.

En otro caso, dado un grafo G, M_f copia sus vértices, copia sus arcos y les asigna costo 1, agrega los arcos que no están en G y les asigna costo 2, y finalmente escribe el número m.

M_f trabaja en tiempo determinístico $O(n^3)$:

- Chequear la validez sintáctica de las entradas tarda $O(n^2)$.
- Escribir V, luego E con costos 1, luego los arcos que no están en E con costos 2, y finalmente m, tarda $O(n^3)$.

Se cumple $G \in HC$ si y sólo si $(G', m) \in TSP$.

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

Fin de Ejemplo

Se prueba fácilmente, construyendo una MTN que trabaja en tiempo polinomial, que $TSP \in NP$. Por lo tanto, considerando el Teorema 8.1, la última reducción es una prueba alternativa a la que vimos en el Ejemplo 7.4 de que $HC \in NP$. No se conoce resolución determinística polinomial para TSP; si existiera valdría que $HC \in P$. El problema del viajante de comercio es tan o más difícil que el problema del circuito de Hamilton, en cuanto a su tiempo de resolución.

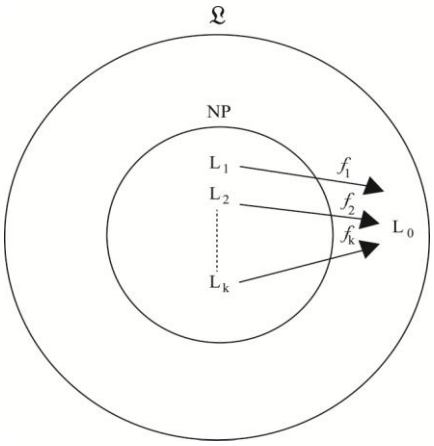
Como veremos enseguida, por medio de las reducciones polinomiales podemos definir una jerarquía temporal dentro de NP. Para estudiar la estructura interna de P, en cambio, hay que recurrir a otro tipo de reducción, al que haremos referencia en la clase siguiente, porque en P siempre se puede reducir polinomialmente un lenguaje a otro. Esto ya lo probamos en R considerando las reducciones generales. Teniendo en cuenta ahora las reducciones polinomiales, se formula lo siguiente. Si L_1 y L_2 son dos lenguajes cualesquiera de P, siendo $L_2 \neq \Sigma^*$ y $L_2 \neq \varnothing$, se cumple que $L_1 \alpha_P L_2$: existe una MT que, a partir de una entrada w perteneciente (no perteneciente) a L_1, lo que puede determinar en tiempo polinomial, genera una salida f(w) perteneciente (no perteneciente) a L_2, lo que puede efectuar en tiempo constante, escribiendo un elemento que está (no está) en L_2.

Habiendo presentado las reducciones polinomiales, podemos ahora introducir el concepto de NP-completitud.

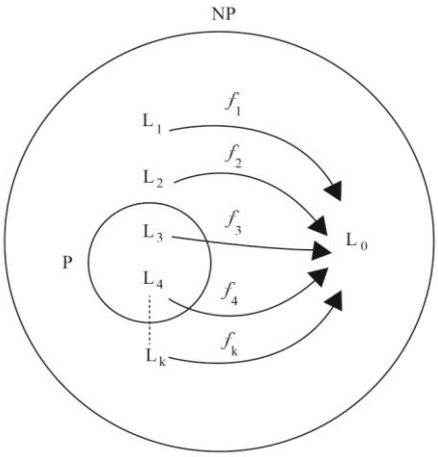
Definición 8.2. Problemas NP-completos

Un lenguaje L_0 es *NP-difícil* (o *NP-hard*, o $L_0 \in NPH$) si y sólo si para todo lenguaje L $\in$ NP se cumple que $L \alpha_P L_0$. En palabras, L_0 es NP-difícil si y sólo si todos los

lenguajes de NP se reducen polinomialmente a él, no importa a qué clase pertenezca, como lo muestra la siguiente figura:



Si en particular L_0 pertenece a NP, se define que es *NP-completo* (o $L_0 \in$ NPC). Ahora la figura correspondiente es la que se muestra a continuación:



Las expresiones lenguaje NP-difícil y problema NP-difícil se usarán en forma indistinta, lo mismo para el caso de las expresiones lenguaje NP-completo y problema NP-completo.

Fin de Definición

Con el siguiente teorema se formaliza que los problemas NP-completos son los más difíciles de NP.

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

La prueba es muy sencilla. Supongamos que un lenguaje NP-completo L_0 está en P. Entonces, si L es algún lenguaje de NP:

- Por ser L_0 un lenguaje NP-completo, se cumple que $L \alpha_P L_0$.
- Y por ser L_0 además un lenguaje de P, se cumple que $L \in P$.

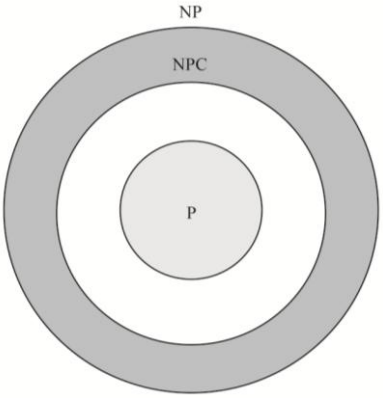
Como esto vale para todo lenguaje L de NP, entonces $NP \subseteq P$, y así: P = NP.

Fin de Teorema

Por lo tanto, asumiendo $P \neq NP$, probar que un problema es NP-completo equivale a "condenarlo" a estar en NP – P. El problema, como cualquier otro de NPC, está entre los más difíciles de NP, lo que se entiende porque resolverlo implica resolver cualquier problema de NP.

Además, demostrando que un problema NP-completo se resuelve en tiempo determinístico T(n), se estaría probando que $NP \subseteq DTIME(T(n^k))$. Por ejemplo, si T(n) = $n^{\log n}$, es decir tiempo no polinomial pero subexponencial, se cumpliría la inclusión de NP en $DTIME(n^{k.\log n})$.

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



Como mostramos en la figura, los problemas de NPC están en la franja de NP "más alejada" de P. Entre NPC y P queda una franja intermedia de problemas, a la que nos referiremos en la clase próxima.

Una vez probado que un problema es NP-completo, se pueden plantear distintas alternativas para tratarlo: analizar el caso promedio, desarrollar algoritmos de *aproximación*, desarrollar algoritmos *probabilísticos*, desarrollar incluso algoritmos exponenciales tratables para instancias de tamaño acotado, etc. En la Clase 10 consideraremos algunos de estos caminos.

Desde la aparición del concepto de NP-completitud hace más de cuarenta años, se han encontrado cientos de problemas NP-completos. Históricamente, el primero fue el problema SAT, de satisfactibilidad de las fórmulas booleanas (sin ninguna restricción de sintaxis). Probamos a continuación que SAT ∈ NPC (Teorema de Cook).

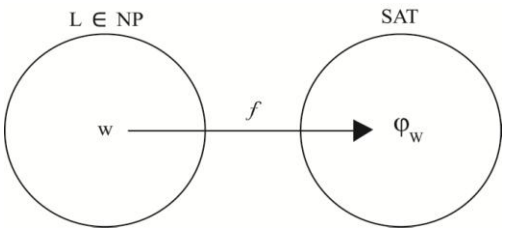
Teorema 8.3. El problema SAT es NP-completo

Sea SAT = {φ | φ es una fórmula booleana satisfactible} el lenguaje que representa el problema SAT. El algoritmo determinístico natural para reconocer SAT, sobre una fórmula de m variables, consiste en probar en el peor caso 2^m asignaciones de valores de verdad, y por lo tanto tarda $O(2^n)$. La prueba de que SAT ∈ NP queda como ejercicio. En lo que sigue demostramos que SAT es NP-difícil, probando que todo lenguaje de NP se reduce polinomialmente a él.

Definición de la función de reducción.

Dado un lenguaje L de NP, tal que M es una MTN que lo reconoce en tiempo polinomial p(n), la idea es transformar toda cadena w en una determinada fórmula booleana φ_w, de modo tal que si alguna computación de M acepta w, entonces existe alguna asignación de valores de verdad que satisface φ_w, y si en cambio todas las computaciones de M rechazan w, entonces no existe ninguna asignación que satisface φ_w.

La figura siguiente ilustra esta idea:



La función de reducción f genera una fórmula φ_w que consiste en la conjunción de cuatro subfórmulas, que enseguida describimos. Antes caben las siguientes aclaraciones:

- Una computación de M a partir de w, se representa mediante una cadena $\#\beta_0\#\beta_1\#\beta_2\ldots\#\beta_{p(n)}$, siendo n = |w|, y β_k la configuración k-ésima de la computación. Si β_k es una configuración final, se hace $\beta_k = \ldots = \beta_{p(n)}$. Se asume, sin perder generalidad, que M tiene una sola cinta.
- En cada β_k, 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 de M), se agrupan en un solo símbolo compuesto. Así, los símbolos para representar una computación de M varían en un alfabeto Ψ formado por ternas con un estado de Q_M, un símbolo de $\Gamma_M \cup \{\#\}$ y un número entre 1 y K, o bien con un blanco, un símbolo de $\Gamma_M \cup \{\#\}$ y otro blanco.
- Como M trabaja en tiempo p(n), entonces no se necesitan más que p(n) símbolos para representar cada β_k. Por lo mismo, la representación de una computación tiene p(n) + 1 configuraciones, y por lo tanto $(p(n) + 1)^2$ símbolos incluyendo los símbolos #.
- Por cada uno de los $(p(n) + 1)^2$ símbolos se va a crear en la fórmula una variable booleana c_{ix}, con i variando entre 0 y $(p(n) + 1)^2 - 1$, y el subíndice x variando en el alfabeto Ψ. Que c_{ix} sea verdadera (falsa) va a representar que el i-ésimo símbolo es (no es) x.
- Para simplificar la escritura, las fórmulas $c_{1x} \wedge c_{2x} \wedge \ldots$, y $c_{1x} \vee c_{2x} \vee \ldots$, se abrevian con $\bigwedge_i c_{ix}$ y $\bigvee_i c_{ix}$, respectivamente. El rango de i se establece en cada caso, mientras que x siempre varía en el alfabeto Ψ.

Se define $\varphi_w = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$, tal que:

La subfórmula φ_1 establece que en una misma posición i no pueden haber dos símbolos distintos x e y del alfabeto Ψ:

$$\varphi_1 = \bigwedge_i [(\bigvee_x c_{ix}) \wedge \neg(\bigvee_{x \neq y} c_{ix} \wedge c_{iy})]$$

con i = 0, …, $(p(n) + 1)^2 - 1$. La subfórmula φ_2 describe la configuración inicial β_0 de M con entrada w, y se expresa a su vez mediante una conjunción de la forma

$\varphi_2 = \varphi_{21} \wedge \varphi_{22} \wedge \varphi_{23} \wedge \varphi_{24}$, con:

$\varphi_{21} = c_{0,\#} \wedge c_{p(n)+1,\#}$

$\varphi_{22} = c_{1,y1} \vee c_{1,y2} \vee \ldots \vee c_{1,ym}$

donde los y_i son todos los símbolos compuestos de Ψ que representan el estado inicial q_0 de M, el primer símbolo de w, y un número posible de próximo paso de M (variando entre 1 y K).

$\varphi_{23} = c_{2,w2} \wedge c_{3,w3} \wedge \ldots \wedge c_{n,wn}$

$\varphi_{24} = c_{n+1,B} \wedge c_{n+2,B} \wedge \ldots \wedge c_{p(n),B}$

La subfórmula φ_3 establece que la última configuración tiene un símbolo compuesto con el estado final q_A:

$$\varphi_3 = \bigvee_i (\bigvee_x c_{ix})$$

tal que i = p(n).(p(n) + 1) + 1, …, $(p(n) + 1)^2 - 1$, y el símbolo x de Ψ incluye el estado q_A. Finalmente, la subfórmula φ_4 establece que β_k es una configuración siguiente posible de β_{k-1}, con k = 1, …, p(n), según la selección no determinística del próximo paso especificada en β_{k-1} y la relación de transición de M:

$$\varphi_4 = \bigwedge_i (\bigvee_{v,x,y,z} c_{i-p(n)-2,v} \wedge c_{i-p(n)-1,x} \wedge c_{i-p(n),y} \wedge c_{iz})$$

tal que i = p(n) + 2, …, $(p(n) + 1)^2 - 1$, y los símbolos v, x, y, z, de Ψ cumplen el predicado S(v, x, y, z), el cual es verdadero si y sólo si z puede aparecer en la posición i de una configuración, estando v, x, y, en las posiciones i – 1, i, i + 1, de la configuración anterior (se debe tener en cuenta particularmente que dos configuraciones consecutivas β_{k-1} y β_k son iguales cuando β_{k-1} tiene un estado final).

<u>La función f es total y pertenece a FP</u>.

Existe una MTD M_f que a partir de una entrada w (y una MTN M), genera una fórmula booleana φ_w tal como la hemos descripto recién. Además, claramente M_f genera φ_w en tiempo determinístico $O(p(n)^2)$.

Se cumple $w \in L$ si y sólo si $\varphi_w \in SAT$.

a. Si w es una cadena de L, entonces existe una computación C de M que la acepta. Por cómo se construye φ_w, asignando valores de verdad a φ_w consistentemente con C se obtiene una evaluación verdadera de la misma, lo que significa que φ_w pertenece a SAT.
b. Si φ_w es una fórmula de SAT, entonces existe una asignación A de valores de verdad que la satisface. Por cómo se construye φ_w, generando las distintas configuraciones β_i especificadas anteriormente consistentemente con A se obtiene la representación de una computación de M que acepta una cadena w, lo que significa que w pertenece a L.

Fin de Teorema

En la primera parte del libro indicamos que L_U está entre los lenguajes más difíciles de RE, porque todos los lenguajes de RE se reducen a él (por eso empleamos el término RE-completo). Se cumple además que L_U es NP-difícil (no es NP-completo porque ni siquiera es recursivo): todo lenguaje de NP se reduce linealmente a L_U. Con esta última consideración se puede definir fácilmente otro lenguaje NP-completo, que presentamos en el siguiente ejemplo. El lenguaje es igual a L_U, salvo que las cadenas incluyen un tercer componente, que es una cota temporal (justamente el lenguaje representa el problema acotado de pertenencia).

Ejemplo 8.3. El problema acotado de pertenencia es NP-completo

Sea $L_{U\text{-}k}$ = {(<M>, w, 1^k) | M es una MTN con una cinta que acepta w en a lo sumo k pasos} el lenguaje que representa el problema acotado de pertenencia. Se prueba que $L_{U\text{-}k}$ es NP-completo. Primero probamos que $L_{U\text{-}k} \in$ NP. La siguiente MTN $M_{U\text{-}k}$ lo reconoce en tiempo polinomial. Dada una entrada v, $M_{U\text{-}k}$ hace:

1. Si v es inválida, rechaza.
2. Genera no determinísticamente un cadena z de tamaño a lo sumo $(k + 1)^2$, con símbolos de <M>, w, etc. (para representar una posible computación de M a partir de w de a lo sumo k pasos, como se hizo en la prueba del Teorema de Cook).

3. Acepta si y sólo si z representa una computación de M que acepta w en a lo sumo k pasos.

Queda como ejercicio probar que $L(M_{U\text{-}k}) = L_{U\text{-}k}$ y que $M_{U\text{-}k}$ trabaja en tiempo no determinístico polinomial. Veamos que $L_{U\text{-}k}$ es NP-difícil. Si $L \in NP$ y M es una MTN con una cinta que reconoce L en tiempo polinomial p(n), sea f una función de reducción de L a $L_{U\text{-}k}$ definida por $f(w) = (<M>, w, 1^{p(|w|)})$. Claramente, f es una función total de FP, y además $w \in L$ si y sólo si $(<M>, w, 1^{p(|w|)}) \in L_{U\text{-}k}$.

Fin de Ejemplo

Reduciendo polinomialmente (por ejemplo) desde SAT, podemos encontrar más lenguajes NP-completos, de una manera similar a cómo fuimos poblando las distintas clases de la jerarquía de la computabilidad. Entonces se emplearon reducciones generales, inicialmente a partir de L_U y HP, "por la negativa" (se fueron encontrando lenguajes no recursivos y no recursivamente numerables). Para poblar NPC, en cambio, utilizaremos las reducciones polinomiales y "por la positiva", como lo fundamenta el siguiente teorema.

Teorema 8.4. Si $L_1 \in NPC$, $L_1 \alpha_P L_2$, y $L_2 \in NP$, entonces $L_2 \in NPC$

En palabras: encontrando una reducción polinomial de un lenguaje L_1 NP-completo a un lenguaje L_2 de NP, se prueba que también L_2 es NP-completo. La prueba es muy simple. Sea L algún lenguaje de NP:

- Como $L_1 \in NPC$, entonces se cumple $L \alpha_P L_1$.
- Como $L_1 \alpha_P L_2$, entonces por propiedad transitiva de α_P se cumple $L \alpha_P L_2$.
- Dado que lo anterior vale para todo lenguale L de NP, entonces L_2 es NP-difícil, y como está en NP, también es NP-completo.

Fin de Teorema

Por ejemplo, el lenguaje CSAT, subconjunto de SAT cuyas fórmulas están en la forma normal conjuntiva, es NP-completo: está en NP, y existe una reducción polinomial de SAT a CSAT. A partir de CSAT se puede probar que también el lenguaje 3-SAT, subconjunto de CSAT cuyas fórmulas tienen tres literales por cláusula, es NP-completo.

En el ejemplo siguiente probamos este último enunciado (recordar que en el Ejemplo 7.3 demostramos que 2-SAT está en P).

Ejemplo 8.4. El problema 3-SAT es NP-completo

La prueba de que 3-SAT está en NP queda como ejercicio. Vamos a demostrar que 3-SAT es NP-difícil, presentando una reducción polinomial de CSAT a 3-SAT.

Definición de la función de reducción.

Dada una fórmula booleana en la forma normal conjuntiva $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_k$, la idea es que la función de reducción f la transforme en otra con tres literales por cláusula de la siguiente manera (si es inválida sintácticamente genera otra fórmula inválida sintácticamente). Si una cláusula φ_i tiene un solo literal ζ, la transforma con dos nuevas variables x_1 y x_2 del siguiente modo:

$$\varphi_i = (\zeta \vee x_1 \vee x_2) \wedge (\zeta \vee x_1 \vee \neg x_2) \wedge (\zeta \vee \neg x_1 \vee x_2) \wedge (\zeta \vee \neg x_1 \vee \neg x_2)$$

Si una cláusula φ_i tiene dos literales ζ_1 y ζ_2, la transforma con una nueva variable x del siguiente modo:

$$\varphi_i = (\zeta_1 \vee \zeta_2 \vee x) \wedge (\zeta_1 \vee \zeta_2 \vee \neg x)$$

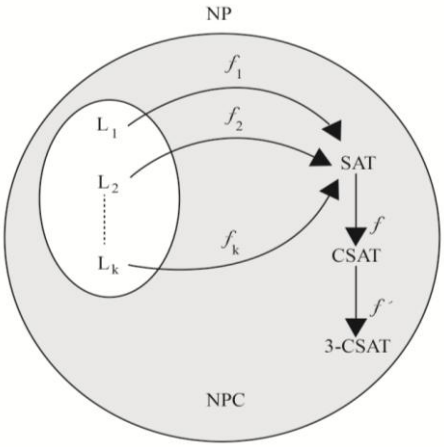
Finalmente, si una cláusula φ_i tiene $m > 3$ literales $\zeta_1, \ldots, \zeta_m$, la transforma con nuevas $m - 3$ variables $x_1, \ldots, x_{m-3}$ del siguiente modo:

$$\varphi_i = (\zeta_1 \vee \zeta_2 \vee x_1) \wedge (\zeta_3 \vee \neg x_1 \vee x_2) \wedge (\zeta_4 \vee \neg x_2 \vee x_3) \wedge \ldots \wedge (\zeta_{m-2} \vee \neg x_{m-4} \vee x_{m-3}) \wedge (\zeta_{m-1} \vee \zeta_m \vee \neg x_{m-3})$$

Queda como ejercicio probar que la función de reducción f es total y pertenece a FP, y que $\varphi \in$ CSAT si y sólo si $f(\varphi) \in$ 3-SAT.

Fin de Ejemplo

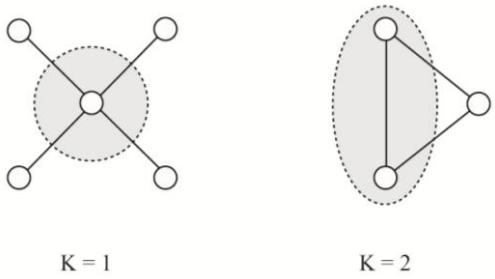
La figura siguiente muestra los primeros representantes de la clase NPC:



La reducción del ejemplo anterior consiste en modificar componentes de fórmulas booleanas para obtener otras fórmulas booleanas con una forma determinada. Lo mismo se puede hacer para reducir SAT a CSAT. Esta técnica se conoce como *reemplazo local*: se detectan componentes básicos en las instancias del problema conocido, y se los modifica uno por uno para producir instancias del problema nuevo. La reducción del ejemplo siguiente es de naturaleza más compleja, se basa en el *diseño de componentes*: de las instancias del problema conocido se construyen conjuntos de componentes adecuadamente interconectados que determinan instancias del problema nuevo. Cada componente cumple una función específica. Así se probó que SAT es NP-completo, generando instancias mediante la conjunción de cuatro subfórmulas booleanas. El siguiente es otro ejemplo de esta técnica, en que se reduce polinomialmente 3-SAT a VC (por *vertex cover* o cubrimiento de vértices), el problema del cubrimiento de vértices, que consiste en determinar si un conjunto de K vértices de un grafo "toca" (cubre) todos sus arcos. De esta manera se prueba que VC es NP-completo.

Ejemplo 8.5. El problema del cubrimiento de vértices es NP-completo

Sea VC = {(G, K) | G es un grafo que tiene un cubrimiento de vértices de tamaño K} el lenguaje que representa el problema del cubrimiento de vértices. Dado un grafo G = (V, E), entonces V' $\subseteq$ V es un cubrimiento de vértices de tamaño K de G, si |V'| = K, e incluye al menos un vértice de todos los arcos de G. La figura siguiente muestra dos cubrimientos de vértices, de tamaños K = 1 y K = 2:



La prueba de que VC está en NP queda como ejercicio. Para probar que es NP-difícil, definimos una reducción polinomial de 3-SAT a VC.

Definición de la función de reducción.

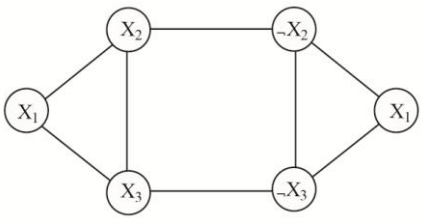
Dada una fórmula booleana φ en la forma normal conjuntiva y con tres literales por cláusula, se define

$$f(\varphi) = (G, 2C)$$

tal que C es la cantidad de cláusulas de φ, y G es un grafo que se construye del siguiente modo:

- Por cada literal de φ se crea un vértice en G.
- Todo par de vértices de G creados a partir de dos literales de una misma cláusula de φ, se unen por un arco. A este enlace lo denominamos de *tipo 1*, como así también a los triángulos resultantes.
- Todo par de vértices de G creados a partir de dos literales x_i y $\neg x_i$ de φ, también se unen por un arco. A este enlace lo denominamos de *tipo 2*.

Por ejemplo, la figura siguiente muestra el grafo que se genera a partir de la fórmula booleana $\varphi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$:



Queda como ejercicio probar que la función de reducción f es total y pertenece a FP.

Se cumple $\varphi \in$ 3-SAT si y sólo si (G, 2C) $\in$ VC.

a. Si φ es una fórmula de 3-SAT, y A es una asignación de valores de verdad que la satisface, entonces al menos un literal de toda cláusula es verdadero. Considerando A, el siguiente conjunto de vértices V' es un cubrimiento de tamaño 2C del grafo G construido. Todo vértice asociado a un literal falso se incluye en V', y si luego de esta inclusión hay triángulos de tipo 1 que no tienen dos vértices en V' (caso de cláusulas con dos o tres literales verdaderos), entonces se agregan a V' vértices cualesquiera de dichos triángulos hasta lograrlo. Así se cumple que |V'| = 2C. También se cumple que V' cubre 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 es verdadero, entonces el literal negado es falso, y así el vértice asociado a este último pertenece a V'.
b. Si (G, 2C) está en VC, y V' es un cubrimiento de 2C vértices del grafo G construido a partir de la fórmula φ, entonces la siguiente asignación de valores de verdad A satisface φ. A los literales asociados a los vértices que no están en V', les asigna el valor verdadero, y al resto les asigna valores consistentes cualesquiera. La asignación A satisface φ porque V' tiene necesariamente dos vértices por triángulo de tipo 1, y así al menos un literal de cada cláusula es verdadero. Por otra parte, A no puede ser inconsistente, no puede suceder que un literal sea verdadero en una cláusula y el literal negado sea verdadero en otra, porque si no, el enlace de tipo 2 asociado a ellos no estaría cubierto por V'.

Fin de Ejemplo

Una tercera técnica, más simple que las anteriores, consiste directamente en reducir un problema a otro similar, como por ejemplo la reducción polinomial que presentamos antes de HC a TSP (Ejemplo 8.2). Se muestra a continuación otra reducción de este tipo, que prueba que el problema del clique (presentado en el Ejemplo 7.5) es NP-completo.

Ejemplo 8.6. El problema del clique es NP-completo

En el Ejemplo 7.5 especificamos el lenguaje que representa el problema del clique: CLIQUE = {(G, K) | G tiene un clique de tamaño K}, y probamos que está en NP.

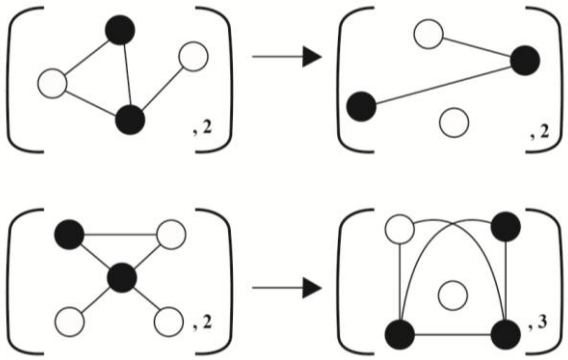
Ahora vamos a demostrar que es NP-difícil, con una reducción polinomial de VC a CLIQUE.

Definición de la función de reducción.

Dado un grafo válido G con m vértices, y un número natural $K \leq m$, definimos la función de reducción

$$f((G, K)) = (G^C, m - K)$$

siendo G^C el grafo "complemento" de G (tiene los mismos vértices que G y sólo los arcos que G no tiene). La figura siguiente muestra dos casos de aplicación de la función de reducción:



En el primer caso, de un grafo con 4 vértices y un cubrimiento de vértices de tamaño 2 (marcado en la figura), se pasa al grafo "complemento", que tiene un clique de tamaño 4 – 2 = 2 (marcado). En el segundo caso, de un grafo con 5 vértices y un cubrimiento de vértices de tamaño 2, se pasa al grafo "complemento", que tiene un clique de tamaño 5 – 2 = 3. Queda como ejercicio probar que la función de reducción f es total y pertenece a FP.

Se cumple $(G, K) \in VC$ si y sólo si $(G^C, m - K) \in CLIQUE$.

Probamos sólo un sentido, el otro se demuestra similarmente y queda como ejercicio. Supongamos que $(G, K) \in VC$, y que V' es un cubrimiento de vértices de G de tamaño K. Veamos que V – V' es un clique de G^C de tamaño m – K, y así que $(G^C, m - K) \in$ CLIQUE.

Por un lado, el conjunto de vértices V – V' tiene tamaño m – K.

Por otro lado, supongamos que G^C no incluye, por ejemplo, el arco (i , h), siendo i y h vértices de V – V'. Entonces (i, h) es un arco de G, siendo i y h vértices que no están en V', por lo que V' no es un cubrimiento de vértices de G (absurdo).

Fin de Ejemplo

Una variante de la última técnica es la *restricción*. Dado un problema nuevo L_2, la idea es mostrar que un problema conocido L_1 es un caso especial de L_2. Se definen restricciones sobre las instancias de L_2 para relacionarlas con las de L_1. Por ejemplo, se puede reducir por restricción el problema CLIQUE al problema del isomorfismo de subgrafos, que consiste en determinar, dados dos grafos G_1 y G_2, si G_1 es isomorfo a un subgrafo de G_2 (ya definimos en el Ejercicio 12 de la clase anterior que dos grafos son isomorfos cuando son iguales salvo por los nombres de sus arcos, que son pares de vértices). Otro ejemplo es la reducción por restricción que se puede definir de HC al problema del circuito de Hamilton pero ahora en grafos orientados. Queda como ejercicio construir ambas reducciones polinomiales.
Considerando los distintos grados de dificultad de lás técnicas mencionadas (que no son todas), una estrategia razonable para probar que un lenguaje L de NP es NP-completo, es encontrar una reducción polinomial de otro problema NP-completo a L:

1. Primero, recurriendo a un problema similar (empleando eventualmente una restricción).
2. Luego, empleando una modificación de componentes.
3. Luego, empleando un diseño de componentes.

Al igual que en P, los lenguajes de NPC están íntimamente relacionados mediante las reducciones polinomiales. Por definición, todo lenguaje NP-completo se reduce polinomialmente a otro. Dada una función de reducción f entre dos lenguajes NP-completos L_1 y L_2, no necesariamente f^{-1} debería ser una función de reducción de L_2 a L_1. De todos modos, se cumple que para todo par de lenguajes NP-completos conocidos L_1 y L_2, existe una función de reducción h de L_1 a L_2, tal que h^{-1} es una función de reducción de L_2 a L_1. Se dice en este caso que L_1 y L_2 son *polinomialmente isomorfos*, o directamente *p-isomorfos*. Más precisamente, dada una reducción polinomial f entre dos lenguajes NP-completos conocidos L_1 y L_2, y una reducción polinomial g entre L_2 y L_1,

se puede construir a partir de ellas, de manera sistemática, una biyección h entre L_1 y L_2, tal que h y h^{-1} se computan en tiempo determinístico polinomial:

1. Primero, con el uso de funciones denominadas de *padding* (o relleno), las funciones f y g se transforman, manteniendo su eficiencia, en funciones f' y g' inyectivas y de longitud creciente.
2. Luego se completa la construcción para lograr la suryectividad. A partir de f' y g' se obtiene una biyección h, tal que h y h^{-1} se computan eficientemente.

Existe una conjetura, la Conjetura de Berman-Hartmanis, que justamente establece que todos los lenguajes NP-completos son p-isomorfos. Si se comprobara la conjetura valdría $P \neq NP$:

- Supongamos por el contrario que P = NP. Entonces P = NPC (el caso $P \subseteq NPC$ se cumple porque todos los lenguajes de P se reducen polinomialmente entre sí).
- Entonces todos los lenguajes finitos, por estar en P, son NP-completos.
- Pero entonces, como en NPC hay lenguajes finitos e infinitos, y no puede haber un p-isomorfismo entre un lenguaje finito y un lenguaje infinito, no todos los lenguajes NP-completos son p-isomorfos (absurdo).

Existe también una contra-conjetura, establecida por Joseph y Young, que enuncia que podrían existir lenguajes NP-completos (no naturales) no p-isomorfos a SAT.
Otra caracterización de la clase NPC relacionada con la anterior se refiere a la densidad de los lenguajes NP-completos: todos los lenguajes NP-completos conocidos son *exponencialmente densos* (o directamente *densos*), lo que significa que la cantidad de sus cadenas de longitud a lo sumo n, para cualquier n, no se puede acotar por un polinomio p(n) (los lenguajes que en cambio cumplen esta propiedad se denominan *polinomialmente densos*, o también *dispersos*). Es interesante observar que si dos lenguajes L_1 y L_2 son p-isomorfos por medio de una función h que se computa en tiempo determinístico polinomial p(n), entonces sus densidades, digamos $dens_1(n)$ y $dens_2(n)$, se relacionan polinomialmente:

- Por un lado, las cadenas de L_1 de longitud a lo sumo n se transforman mediante h en cadenas de L_2 de longitud a lo sumo p(n). Como h es inyectiva, entonces $dens_1(n) \leq dens_2(p(n))$.
- Por otro lado, si h^{-1} se computa en tiempo determinístico polinomial q(n), entonces por la misma razón se cumple que $dens_2(n) \leq dens_1(q(n))$.

De esta manera, ningún lenguaje NP-completo conocido puede ser p-isomorfo a un lenguaje disperso. Más aún, se prueba que la existencia de un lenguaje NP-completo disperso implica P = NP (intuitivamente, los lenguajes más difíciles de NP no pueden tener "pocas" cadenas).

Como veremos en las próximas dos clases, la completitud es aplicable a cualquier clase de complejidad, no solamente NP. Es un concepto central en la teoría de la complejidad computacional, al extremo de considerarse que el significado de un problema se entiende definitivamente cuando se prueba que es completo en una determinada clase, y que una clase no es importante en la práctica si no tiene problemas naturales completos.

Ejercicios de la Clase 8

1. Probar que las reducciones polinomiales de problemas son reflexivas y transitivas.
2. Una reducción polinomial no determinística f se define como una determinística, con el agregado de que alguna computación de la MTN asociada debe calcular f(w) a partir de la entrada w. Indicar si en este caso siguen valiendo la reflexividad y la transitividad.
3. Completar la prueba del Teorema 8.1.
4. Dados dos lenguajes A y B distintos de $\emptyset$ y Σ^*, determinar si se cumple:
   i. Si $B \in P$, entonces $A \cap B \alpha_P A$.
   ii. Si $B \in P$, entonces $A \cup B \alpha_P A$.
5. Probar que los lenguajes HP y L_U pertenecen al conjunto NPH.
6. Probar que si $L_1 \alpha_P L_2$, $L_2 \alpha_P L_1$, y $L_1 \in NPC$, entonces $L_2 \in NPC$.
7. Completar la prueba del Teorema 8.3.
8. Completar la prueba del Ejemplo 8.3.
9. Completar la prueba del Ejemplo 8.4.
10. Completar la prueba del Ejemplo 8.5.
11. Completar la prueba del Ejemplo 8.6.

12. Dados dos lenguajes A y B distintos de ∅ y Σ*, tales que A ∈ NP y B ∈ P, determinar si se cumple:
    i. Si A ⋂ B es NP-completo, entonces A es NP-completo.
    ii. Si A ⋃ B es NP-completo, entonces A es NP-completo.
13. Probar que los siguientes lenguajes o problemas son NP-completos. En cada caso identificar además la naturaleza del problema que dificulta encontrar un algoritmo determinístico polinomial:
    i. IND = {(G, K) | K es un número natural y G = (V, E) es un grafo que tiene un conjunto independiente de K vértices}. Un conjunto de vértices V' ⊆ V de un grafo G es independiente, si y sólo si todo par de vértices s, t de V' cumple que (s, t) ∉ E.
    ii. HPA = {(G, s, t) | G es un grafo que tiene un camino de Hamilton del vértice s al vértice t}. Un grafo G tiene un camino de Hamilton de s a t si tiene un camino de s a t que recorre los vértices restantes una sola vez.
    iii. El lenguaje de las fórmulas booleanas en la forma normal disyuntiva que tienen una asignación de valores de verdad que no las satisfacen.
    iv. Dado un grafo con un circuito de Hamilton, determinar si tiene otro circuito de Hamilton.
    v. Dado un conjunto de n conjuntos, determinar si existen K conjuntos, con K < n, cuya unión coincide con la unión de los n conjuntos.
    vi. Dado un grafo G = (V, E) y un número natural K, determinar si existe un conjunto de vértices V' ⊆ V, con |V'| < K, tal que todo circuito de G incluye al menos un vértice de V'.
    vii. Dado un grafo G = (V, E), determinar si G tiene un conjunto independiente V', tal que para todo vértice v ∈ V – V' existe al menos un arco entre v y algún vértice de V'.
    viii. El problema 3-SAT restringido a que exactamente un literal por cláusula es verdadero.
    ix. El problema 3-SAT restringido a que toda cláusula tiene tres variables distintas.
    x. El problema CSAT restringido a que toda cláusula es una cláusula de Horn o tiene dos literales.
14. Probar que MCLIQUE = {(G, L, K) | el grafo G tiene L cliques de tamaño K} es NP-difícil. ¿Se cumple que MCLIQUE ∈ NPC?

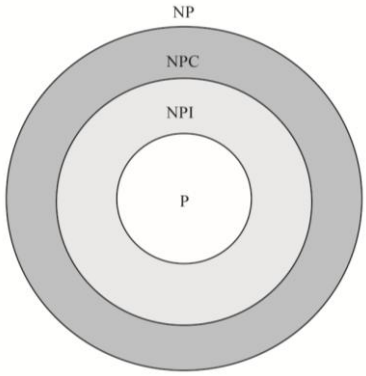
15. Probar que 3-SAT es NP-completo, mediante una reducción polinomial directa desde SAT.
16. Probar que CLIQUE es NP-completo, mediante una reducción polinomial directa desde CSAT.
17. Probar que se puede reducir por restricción:
    i. CLIQUE al problema del isomorfismo de subgrafos.
    ii. HC al problema del circuito de Hamilton en grafos orientados.
18. Si P ≠ NP, probar que no es decidible determinar, dado un lenguaje L de NP, si L está en P.

Clase 9. Otras clases de complejidad

Hasta el momento hemos descripto con bastante detalle, en el marco de la jerarquía temporal y asumiendo P ≠ NP, la clase de problemas NP y sus subclases P y NPC.
En esta clase presentamos de una manera muy general otras clases de la jerarquía temporal. También presentamos los aspectos más relevantes de la complejidad espacial y la jerarquía asociada, que solapamos con la jerarquía temporal para mostrar distintas relaciones entre clases de problemas de las dos jerarquías.

9.1. LA CLASE NPI

Asumiendo P ≠ NP, se prueba que NP incluye, además de P y NPC, una tercera subclase de problemas, NPI, así llamada por incluir problemas de dificultad intermedia comparados con los de las otras dos subclases. La figura siguiente ilustra la estructura interna de NP con esta nueva franja de problemas:



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

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

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

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

Por ejemplo, haciendo B = SAT, entonces existe un lenguaje D de fórmulas booleanas reconocibles polinomialmente, tal que SAT restringido a D no es ni NP-completo ni está en P.

Utilizando la misma idea de construcción de antes, partiendo ahora de algún lenguaje A_1 perteneciente a NPI (cumple que está en EXP – P), se puede obtener un subconjunto A_2 perteneciente a NPI más fácil que A_1 (A_2 α_P A_1 y no vale A_1 α_P A_2). Del mismo modo se puede obtener A_3 a partir de A_2, y así sucesivamente. Es decir, se puede definir en NPI una jerarquía infinita de problemas de dificultad intermedia a partir de A_1. Y también otra, arrancando de otro lenguaje inicial A'_1 de NPI, y otra a partir de A''_1, etc.

Otro resultado que se obtiene del teorema es la existencia en NPI de pares de lenguajes incomparables, es decir pares de lenguajes tales que ninguno se reduce polinomialmente al otro.

Existen problemas naturales en NP a los que no se les han encontrado resoluciones eficientes, ni tampoco pruebas de su pertenencia a la clase NPC, por lo que son candidatos a estar en NPI. Un ejemplo es el problema del isomorfismo de grafos. Notar que en la clase pasada indicamos que el problema del isomorfismo de subgrafos es NP-completo. Esta diferencia se relaciona con el hecho de que las instancias de los problemas NP-completos tendrían cierta redundancia de información que los problemas no NP-completos no tendrían. En el caso del isomorfismo podría explicarse de la siguiente manera: extraer o agregar arcos fuera de un subgrafo de un grafo G_2 isomorfo a un grafo G_1 no impacta sobre el isomorfismo, pero en cambio cualquier alteración sí impacta cuando el isomorfismo se establece considerando la totalidad de los grafos G_1 y

G_2. Por otro lado, la falta de redundancia en sus instancias no le sería suficiente al problema de los grafos isomorfos para estar en P. Hay otros problemas de isomorfismos (de grupos, subgrupos, etc.) que también son candidatos a pertenecer a NPI.

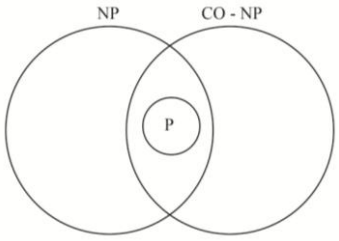
La factorización es otro problema candidato para estar en NPI. El problema de decisión asociado consiste en determinar si un número natural n tiene un factor menor que un número natural m. A pesar de que los problemas complementarios de primalidad y composicionalidad están en P (se demostró una década atrás), la factorización parece ser más difícil, sin llegar al extremo de la NP-completitud. En este caso, una explicación de por qué no sería tan difícil es que la factorización también está en CO-NP. La relación entre la NP-completitud y la pertenencia a NP $\cap$ CO-NP se trata en la sección siguiente (por lo pronto podemos comentar que, intuitivamente, teniendo más información sobre un problema de NP al saber que además está en CO-NP, el problema se torna menos difícil). Otra causa de la dificultad menor de la factorización, considerándola como función, sería que siempre tiene solución, por el Teorema Fundamental de la Aritmética (a modo de ilustración, comparémosla por ejemplo con la función que obtiene una asignación de valores de verdad que satisface una fórmula booleana: esta función, asociada al problema NP-completo SAT, no siempre tiene solución).

Una interesante caracterización de NPI se relaciona con la densidad de sus lenguajes. Vimos que asumiendo P $\neq$ NP, los lenguajes NP-completos no pueden ser dispersos. En cambio es plausible la existencia de este tipo de lenguajes en NPI.

9.2. LA CLASE CO-NP

Ya hemos hecho referencia al conjunto CO-NP en la Clase 7. CO-NP agrupa, dentro de EXP, a los lenguajes complemento de los lenguajes de NP. Mencionamos en la Clase 7 como ejemplo de lenguaje en CO-NP a NOCLIQUE, el complemento de CLIQUE. Enunciamos que P $\subseteq$ NP $\cap$ CO-NP, y también destacamos que se conjetura que NP $\neq$ CO-NP. Notar que la asunción NP $\neq$ CO-NP es más fuerte que la asunción P $\neq$ NP: si P = NP, como P es cerrada con respecto al complemento entonces también lo es NP, por lo que vale NP = CO-NP.

La figura siguiente ilustra la relación entre las clases NP, CO-NP y P, asumiendo NP $\neq$ CO-NP:



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

Un problema de CO-NP es CO-NP-completo si todos los problemas de CO-NP se reducen polinomialmente a él. Se demuestra fácilmente que si un problema es NP-completo, entonces su complemento es CO-NP-completo (la prueba queda como ejercicio). Un importante representante de los problemas CO-NP-completos es el problema de la validez de las fórmulas booleanas, que consiste en determinar si una fórmula booleana es satisfactible por toda asignación de valores de verdad (recordemos que en el Ejemplo 4.8 probamos que el problema es indecidible considerando las fórmulas de la lógica de primer orden). Identificando con BVAL al lenguaje que representa este problema, se prueba fácilmente que BVAL es CO-NP-completo:

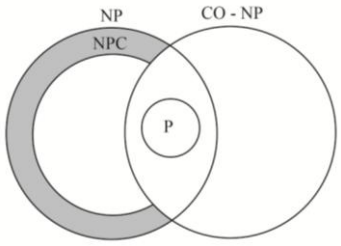
- BVAL está en CO-NP: toda fórmula booleana de BVAL tiene una descalificación suscinta (una asignación de valores de verdad que no la satisface).
- BVAL es CO-NP-difícil: NOSAT (el problema complemento de SAT) es CO-NP-completo, y se reduce polinomialmente a BVAL mediante la función de reducción $f(\varphi)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

Asumiendo NP ≠ CO-NP, se prueba que NPC y NP ⋂ CO-NP son disjuntos. Intuitivamente, como ya comentamos en la sección previa, se tiene más información en los problemas de NP ⋂ CO-NP que en los problemas de NP – (NP ⋂ CO-NP). Y al ser NPC la subclase de los problemas más difíciles de NP, entonces los problemas de NP ⋂ CO-NP deberían ser de dificultad baja (los de P) o intermedia (los de NPI). Formalmente, si L es un lenguaje NP-completo y además está en CO-NP, entonces se

cumple que NP = CO-NP. Probamos a continuación el caso NP $\subseteq$ CO-NP, y la demostración de CO-NP $\subseteq$ NP queda como ejercicio:

- Dado un lenguaje L' perteneciente a NP, se cumple L' α_P L.
- Entonces también se cumple L'^{C} α_P L^{C}.
- Como $L^{C} \in$ NP, entonces $L'^{C} \in$ NP, o lo que es lo mismo, L' $\in$ CO-NP.

La figura siguiente ilustra el último resultado:



Notar que toda instancia de un problema de NP $\cap$ CO-NP tiene o bien un certificado suscinto, cuando la instancia es positiva, o bien una descalificación suscinta, cuando la instancia es negativa. La naturaleza de una y otra clase de cadenas pueden ser muy distintas, como así también los algoritmos para chequearlas.
La complejidad descriptiva (recordar lo que vimos al final de la Clase 7) también permite caracterizar en el marco de la lógica a la clase CO-NP. Mientras NP coincide con los problemas que pueden especificarse mediante la lógica existencial de segundo orden, CO-NP coincide con los problemas que pueden especificarse mediante la lógica universal de segundo orden.

9.3. TIEMPO EXPONENCIAL

Hemos tratado hasta ahora con clases de complejidad temporal definidas en términos de MT, determinísticas o no determinísticas, que trabajan en tiempo polinomial. Y nos hemos enfocado en un esquema según el cual, probando que un problema es NP-completo, se conjetura que el mismo no tiene resolución eficiente. En los ejemplos siguientes, en cambio, mostramos problemas que efectivamente no tienen resolución eficiente.

En estos casos se consideran las clases de problemas

$$PEXP = DTIME(2^{p(n)})$$
$$NPEXP = NTIME(2^{p(n)})$$

siendo p(n) un polinomio, y también clases de orden superior como 2-PEXP (tiempo doble exponencial), donde el exponente del 2 no es un polinomio p(n) sino $2^{p(n)}$. La jerarquía exponencial continúa con 3-PEXP, 4-PEXP, etc., y lo mismo se define en términos de tiempo no determinístico. Los ejemplos son:

- El problema de las expresiones regulares con exponenciación: consiste en determinar si una expresión regular con exponenciación denota todas las cadenas de un alfabeto. Ya hemos descripto la sintaxis de las expresiones regulares, sin exponenciación, en el Tema 5.3. El nuevo operador se representa con el símbolo ↑ y se utiliza de la siguiente manera: a↑i denota una cadena de i símbolos a. Se prueba que el problema no puede resolverse en menos de espacio determinístico exponencial, y por lo tanto tampoco en menos de tiempo determinístico exponencial, porque $2^{p(n)}$ celdas no pueden recorrerse en menos de $2^{p(n)}$ pasos. Dado cualquier lenguaje L reconocido por una MTD que trabaja en espacio $2^{p(n)}$, se cumple que L es polinomialmente reducible al lenguaje que representa el problema. Se demuestra además que el problema se resuelve en espacio determinístico exponencial. Otro problema clásico relacionado con las expresiones regulares es el problema de las expresiones regulares equivalentes, que consiste en determinar si dos expresiones regulares son equivalentes, es decir si definen el mismo lenguaje. Distintas variantes de este problema habitan la franja más difícil de la jerarquía temporal.
- El problema de decisión en la teoría de los números reales con adición (sin multiplicación): consiste en determinar la verdad o falsedad de una fórmula de dicha teoría. Se prueba que este problema se resuelve en tiempo determinístico doble exponencial, y en no menos de tiempo no determinístico exponencial. Dado cualquier lenguaje L reconocido por una MTN que trabaja en tiempo $2^{p(n)}$, se cumple que L es polinomialmente reducible al lenguaje que representa el problema. La teoría de los números reales es decidible aún con la multiplicación, lo que contrasta con la indecidibilidad en la teoría de números (es interesante

destacar que el distinto comportamiento de los reales y enteros se observa consistentemente en distintos niveles; por ejemplo, recordar que en la Clase 7 indicamos que la programación lineal está en P, mientras que la programación lineal entera es un problema NP-completo). La teoría de números sin la multiplicación (Aritmética de Presburger), en cambio, es decidible. En este caso, el tiempo mínimo de resolución es aún mayor que en los números reales sin multiplicación: no determinístico doble exponencial.

- El problema de satisfactibilidad en una lógica de primer orden particular: hemos indicado en la Clase 4 que la satisfactibilidad en la lógica de primer orden no es decidible, pero se prueba que sí lo es, restringiendo la sintaxis a fórmulas sin símbolos de función ni igualdad, en la forma prenex y enunciando primero los cuantificadores existenciales y luego los universales (lo que se conoce como *forma de Schönfinkel-Bernays*). En este caso, el problema es completo en la clase de tiempo no determinístico exponencial.
- Problemas sobre grafos representados suscintamente: volvemos por un momento a los problemas sobre grafos, en particular a los que se relacionan con el diseño de circuitos integrados. En este contexto, es requisito primordial la representación suscinta de la información, por lo que se emplean sofisticadas técnicas de codificación. La representación suscinta hace que distintos problemas sobre grafos estudiados previamente varíen ahora dentro del rango de tiempo exponencial, determinístico y no determinístico.

9.4. COMPLEJIDAD ESPACIAL

Definiciones básicas

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

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

similar se define una MT que trabaja en espacio O(S(n)). Cuando $S(n) \geq n$ se puede utilizar directamente una MT con una cinta de entrada común.

Un problema (o lenguaje) pertenece a la clase DSPACE(S(n)), si y sólo si existe una MTD con una cinta de entrada de sólo lectura y varias cintas de trabajo que lo resuelve (o reconoce) en espacio O(S(n)). La misma definición vale para la clase NSPACE(S(n)) considerando las MTN.

La jerarquía espacial está incluida, al igual que la temporal, en la clase R. Si bien en las definiciones anteriores no se explicita que las MT se detienen siempre, se prueba que si existe una MT que trabaja en espacio S(n), entonces existe una MT equivalente que trabaja en el mismo espacio y se detiene siempre. La prueba se basa en que en espacio acotado sólo puede haber un número finito de configuraciones distintas de una MT a partir de una entrada (hemos presentado una prueba de este tipo en el Ejemplo 4.11).

Simplificando como en la jerarquía temporal, asumiremos que lo no polinomial es exponencial, y hablaremos de ahora en más de EXPSPACE (por espacio exponencial) en lugar de R.

Las MT estándar empleadas para el estudio de la complejidad espacial son las MTD con varias cintas. Según la simulación desarrollada en la primera parte del libro (ver Ejemplo 1.5), el espacio consumido es el mismo utilizando MT con cualquier cantidad finita de cintas. De esta manera no se pierde generalidad considerando MT con una sola cinta de trabajo en las definiciones y demostraciones.

Volvemos una vez más al problema de reconocimiento de palíndromes (esta vez con un separador en la mitad), para mostrar un ejemplo de MT que trabaja en espacio menor que lineal.

Ejemplo 9.1. El lenguaje de las cadenas wcw^R está en DSPACE(log n)

Se prueba que $L = \{wcw^R \mid w \in \{a, b\}^*\}$, siendo w^R la cadena inversa de w, pertenece a la clase DSPACE(log n). Sea la siguiente MTD M, que a partir de una entrada v trabaja de la siguiente manera:

1. Escribe la cantidad i de símbolos de la parte izquierda de v, sin incluir el símbolo c, en la cinta de trabajo 1, y la cantidad k de símbolos de la parte derecha de v, sin incluir el símbolo c, en la cinta de trabajo 2. Rechaza si encuentra símbolos distintos de a, b, c, o si no encuentra exactamente un símbolo c. Acepta si $i = k = 0$, y rechaza si $i \neq k$.

2. Compara el símbolo i de la parte izquierda de v, con el símbolo k – i + 1 de la parte derecha de v, y si son distintos rechaza. Si no, hace i := i – 1 en la cinta de trabajo 1. Si i = 0 acepta, y en caso contrario vuelve al paso 2.

Por ejemplo, si al empezar el paso 2 se cumple i = k = 5, M primero compara el quinto símbolo de la parte izquierda de v con el primer símbolo de la parte derecha de v, después el cuarto símbolo de la parte izquierda con el segundo símbolo de la parte derecha, después el tercero de la izquierda con el tercero de la derecha, etc. Claramente, se cumple que L(M) = L (la prueba queda como ejercicio).

M trabaja en espacio determinístico O(log n).

- Las operaciones para calcular, escribir y modificar los contadores i y k consumen espacio O(log |v|) (los números, como siempre, se representan en notación no unaria).
- La comparación de símbolos requiere tener en memoria sólo dos, lo que consume espacio constante.

Por lo tanto, el lenguaje L pertenece a la clase DSPACE(log n).

Fin de Ejemplo

La clase DSPACE(log n) se conoce también como DLOGSPACE. La técnica utilizada en el ejemplo es típica de la complejidad espacial. Se utilizan contadores que representan posiciones de los cabezales, los contadores se representan en bases adecuadas, en las cintas de trabajo se guardan sólo pequeñas subcadenas de la entrada, etc. El siguiente es otro ejemplo en el que se observan estas características. Se refiere al problema de la alcanzabilidad en un grafo, ya mencionado antes. En este caso lo vamos a considerar en grafos orientados.

Ejemplo 9.2. El problema de la alcanzabilidad en grafos orientados está en NSPACE(log n)

Sea O-ALCANZABILIDAD el lenguaje que representa el problema de la alcanzabilidad en un grafo orientado. Se define O-ALCANZABILIDAD = {(G, v_1, v_2) | G es un grafo orientado y existe un camino en G del vértice v_1 al vértice v_2}. Vamos a

probar que el lenguaje está en NSPACE(log n). La siguiente MTN M, con un contador c en la cinta de trabajo 3 que al comienzo vale 1, trabaja de la siguiente manera a partir de una entrada w (como siempre, se asume que la cantidad de vértices de un grafo es m):

1. Si w no es una entrada válida, rechaza.
2. Hace x := v_1 en la cinta de trabajo 1.
3. Escribe no determinísticamente un vértice z de G en la cinta de trabajo 2.
4. Si (x, z) no es un arco de G, rechaza. Si z = v_2, acepta.
5. Hace c := c + 1 en la cinta de trabajo 3, y si c = m, rechaza.
6. Hace x := z en la cinta de trabajo 1, y vuelve al paso 3.

Se cumple O-ALCANZABILIDAD = L(M).
Sólo si existe un camino de v_1 a v_2 en el grafo G, la MTN M lo irá recorriendo vértice a vértice en la cinta de trabajo 2. En el peor caso, M hace m – 1 iteraciones, hasta aceptar en el paso 4 o rechazar en el paso 5.

M trabaja en espacio no determinístico O(log n).
La validación sintáctica en el paso 1 se puede hacer en espacio logarítmico (queda como ejercicio). Con respecto a los pasos 2 a 6, M no construye el camino buscado (esto consumiría espacio lineal), sino que le alcanza con tener en memoria sólo un par de vértices, lo que consume espacio O(log n), con |w| = n. El contador c también ocupa espacio O(log n).

Fin de Ejemplo

Cabe destacar que la alcanzabilidad en grafos no orientados se puede resolver en espacio determinístico logarítmico. La clase NSPACE(log n) se conoce también como NLOGSPACE. Otro par de clases de complejidad que se distinguen en la jerarquía espacial 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. Es decir:

$$PSPACE = \bigcup_{i \geq 0} DSPACE(n^i)$$

$$NPSPACE = \bigcup_{i \geq 0} NSPACE(n^i)$$

Como en el caso del tiempo, se trabaja con funciones S(n) de "buen comportamiento", denominadas ahora *espacio-construibles*. Cumplen que para toda entrada w, con |w| = n, existe una MT que trabaja en espacio determinístico exactamente S(n). Considerando espacio determinístico (pero lo mismo aplica al espacio no determinístico), características relevantes de la jerarquía espacial son:

- El salto de una clase espacial a otra que la incluya estrictamente también se produce por medio de una función mayor que lo determinado por factores constantes, lo que se formaliza mediante un teorema conocido como Teorema de Compresión Lineal (*Linear Tape Compression Theorem*).
- Para que una clase DSPACE($S_2(n)$) incluya estrictamente a una clase DSPACE($S_1(n)$), además de $S_1(n) = O(S_2(n))$ debe darse que $S_2(n)$ sea significativamente mayor que $S_1(n)$ cuando n tiende a infinito. Ahora no aparece el factor logarítmico necesario en la jerarquía temporal (ver Teorema 6.1), porque en la prueba por diagonalización asociada no se presenta el problema de la cantidad de cintas de las MT simuladas. De esta manera, en particular se cumple que DLOGSPACE $\subset$ DSPACE(n^k) para todo número natural k mayor que cero, y por lo tanto

  $$\text{DLOGSPACE} \subset \text{PSPACE}$$

  La prueba de esta inclusión queda como ejercicio.
- También la jerarquía espacial es densa: siempre hay un lenguaje recursivo por fuera de DSPACE(S(n)), siendo S(n) una función total computable.
- Existe una función total computable S(n) tal que DSPACE(S(n)) = PSPACE, por aplicación del ya referido Teorema de la Unión, ahora en el marco de la complejidad espacial. De esta manera se prueba que la clase PSPACE está incluida estrictamente en EXPSPACE.

Relación entre la jerarquía espacial y la jerarquía temporal

Un par de principios básicos que vinculan el tiempo y el espacio consumidos por una MT son los siguientes:

- Si una MT M tarda tiempo T(n), no recorre más de T(n) celdas. Esto corresponde al peor caso, en que M siempre va a la derecha o siempre va a la izquierda. Como caso particular, tiempo determinístico T(n) puede acotarse con espacio determinístico $O(\sqrt{T(n)})$, cuando $T(n) \geq n^2$ y es tiempo-construible, $\sqrt{T(n)}$ es espacio-construible, y se consideran sólo MTD con una cinta. La prueba de esta relación se basa en la técnica de las *secuencias de cruces*, que consiste en simular uno a uno fragmentos de cinta reusando el mismo espacio, teniendo en cuenta las secuencias de estados que se producen en sus delimitaciones.
- En espacio S(n), una MT no puede ejecutar más de $c^{S(n)}$ pasos sin repetir alguna configuración, siendo c una constante que depende de la MT.

Considerando el segundo item se puede concluir que los problemas de DLOGSPACE son tratables, porque se resuelven en tiempo determinístico polinomial. Efectivamente, se cumple que

$$\text{DLOGSPACE} \subseteq \text{P}$$

porque 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 pueden caracterizar por ser problemas de P con resoluciones que consisten, básicamente, en validaciones de propiedades sobre componentes de sus instancias. Probaremos después que los problemas de NLOGSPACE también son tratables, es decir que

$$\text{NLOGSPACE} \subseteq \text{P}$$

Por definición, DLOGSPACE $\subseteq$ NLOGSPACE, y se conjetura que P incluye a ambas clases estrictamente.

A diferencia de lo que sucede con el tiempo, pasar del no determinismo al determinismo en el espacio impacta sólo en un orden cuadrático, lo que se formula en el Teorema de Savitch, que establece que si $L \in \text{NSPACE}(S(n))$, siendo S(n) una función espacio-construible que cumple $S(n) \geq \log n$, entonces $L \in \text{DSPACE}(S^2(n))$. La demostración

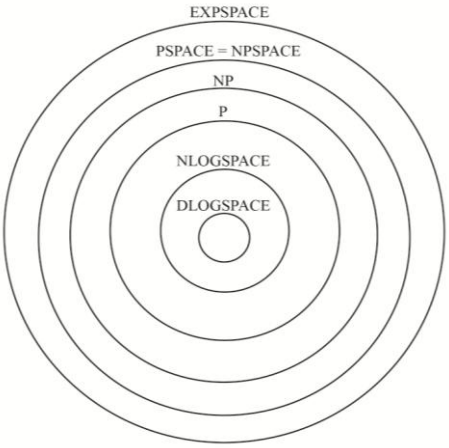
del teorema se basa en una técnica típica de la complejidad espacial, conocida como *método de alcanzabilidad* (porque se refiere a la búsqueda de un camino en un grafo).
Si M es una MTN que a partir de una entrada w trabaja en espacio S(n), sus configuraciones se representan en espacio O(S(n)) por 4-tuplas que contienen la posición del cabezal en la cinta de entrada, la posición del cabezal en la cinta de trabajo, el estado corriente, y el contenido de la cinta de trabajo. Se van generando posibles configuraciones de aceptación C_1, C_2, etc., de la MTN M, y se chequea cada vez si se alcanzan a partir de la configuración inicial C_0 en una cantidad de pasos máxima de $c^{S(n)}$, siendo c una constante que depende de M. Si se alcanza alguna C_i entonces se acepta, y en caso contrario se rechaza. El chequeo de alcanzabilidad desde C_0 a cada C_i se efectúa mediante un procedimiento recursivo que permite, reutilizando espacio, no efectuar más de O(S(n)) invocaciones que ocupan no más de O(S(n)) celdas cada una, llegando así al impacto cuadrático referido. Se requiere que $S(n) \geq \log n$, por la representación de la posición del cabezal en la cinta de entrada. La generación de las distintas configuraciones de tamaño O(S(n)) es factible, por ser S(n) espacio-construible (esta propiedad hace posible definir cadenas de exactamente S(n) celdas). Por el Teorema de Savitch, $NSPACE(n^k) \subseteq DSPACE(n^{2k})$ para todo $k \geq 1$, y así $NPSPACE \subseteq PSPACE$. Por lo tanto, como por definición vale $PSPACE \subseteq NPSPACE$, se cumple

$$PSPACE = NPSPACE$$

En palabras, en la complejidad espacial la clase de los problemas con resolución no determinística polinomial coincide con la clase de los problemas con resolución determinística polinomial. Como $NP \subseteq NPSPACE$ (toda computación con un número polinomial de pasos no ocupa más que un número polinomial de celdas), entonces también se cumple

$$NP \subseteq PSPACE$$

En la figura siguiente se presenta una sola jerarquía con las relaciones mencionadas entre distintas clases de complejidad temporal y espacial:



Se cumplen entonces las siguientes relaciones (falta probar que NLOGSPACE $\subseteq$ P):

$$\text{DLOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{P} \subseteq \text{NP} \subseteq \text{PSPACE}$$

y es un problema abierto determinar cuáles de estas inclusiones son estrictas. Como DLOGSPACE está incluido estrictamente en PSPACE, entonces al menos una tiene que serlo.

Otra diferencia con la complejidad temporal es que en el caso del espacio, toda clase espacial no determinística es cerrada con respecto a la operación de complemento (notar el contraste con la jerarquía temporal, por ejemplo con la conjetura NP $\neq$ CO-NP). Esta propiedad se formula en el Teorema de Immerman, que establece que para toda función espacio-construible $S(n) \geq \log n$ se cumple que NSPACE(S(n)) = CO-NSPACE(S(n)). La demostración también se basa en el método de alcanzabilidad. Primero se prueba que el número C de configuraciones alcanzables por una MTN M, desde una entrada w, que trabaja en espacio S(n), se puede calcular en el mismo espacio no determinístico S(n). Y después se demuestra que conociendo C, se puede aceptar el complemento de L(M) también en espacio no determinísico S(n). Los teoremas de Savitch e Immerman revelan que el impacto del no determinismo es mayor en la complejidad temporal que en la espacial.

Completitud en la jerarquía espacio-temporal

La NP-completitud es un caso particular del concepto de completitud en cualquier clase 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)

significa que no pertenece a P, a menos que P = NP. El mismo criterio se puede aplicar a cualquier clase de complejidad, haciendo referencia a un tipo determinado de reducción.

A propósito, otro tipo de reducción útil para considerar en este contexto es la *reducción logarítmica* (en espacio), que es una m-reducción entre dos lenguajes (o problemas), computable en espacio determinístico logarítmico. En este caso, ni la cinta de entrada ni la cinta de salida de la MT que computa la reducción, intervienen en el cálculo del espacio ocupado. Utilizaremos la expresión $L_1 \ \alpha_{log} \ L_2$ para denotar que existe una reducción logarítmica de L_1 a L_2. Y para simplificar la nomenclatura, emplearemos los términos *poly-time* y *log-space* para referirnos a las reducciones polinomiales y logarítmicas, respectivamente.

Claramente, toda reducción *log-space* es una reducción *poly-time* (queda como ejercicio). De hecho, por ejemplo la reducción utilizada en el Teorema de Cook para probar que SAT es NP-completo es *log-space*: la fórmula booleana construida es lo suficientemente simple como para que la MT M_f que la genera sólo haga uso fundamentalmente del espacio necesario para contar hasta $(p(n) + 1)^2$, que es O(log n).

Además, la relación α_{log} es reflexiva y transitiva. En este último caso, no sirve como prueba la composición de dos MT M_1 y M_2 que trabajan en espacio determinístico logarítmico, porque a partir de una entrada w, con |w| = n, la salida de M_1 puede medir $O(n^k)$, con k constante. Esta dificultad técnica se resuelve del siguiente modo: M_1 no escribe toda la salida, sino que le pasa a M_2 un símbolo por vez, cuando M_2 lo necesita.

Otra propiedad de α_{log} es que las clases DLOGSPACE, NLOGSPACE, P, NP y PSPACE son cerradas con respecto a las reducciones *log-space*, es decir que en cada una de estas clases C vale que si $L_1 \ \alpha_{log} \ L_2$ y $L_2 \in C$, entonces $L_1 \in C$ (ya demostramos antes que P y NP son cerradas con respecto a las reducciones *poly-time*).

Se define que un lenguaje (o problema) L es *C-difícil* con respecto a las reducciones *poly-time* (respectivamente *log-space*) si y sólo si para todo lenguaje (o problema) L' de la clase C se cumple que $L' \ \alpha_P \ L$ (respectivamente $L' \ \alpha_{log} \ L$). Si L además está en la clase C, entonces es *C-completo*.

Con estas definiciones y enunciados podemos generalizar el razonamiento que empleamos antes para analizar en particular la relación entre P y NP: ante la sospecha de que dos clases C_1 y C_2 cumplen que $C_1 \subset C_2$, entonces probar que un problema es C_2-completo significa que no pertenece a C_1, a menos que $C_1 = C_2$. Por ejemplo:

- Si L es NLOGSPACE-completo con respecto a las reducciones *log-space*, y L está en DLOGSPACE, entonces DLOGSPACE = NLOGSPACE.
- Si L es P-completo con respecto a las reducciones *log-space*, y L está en DLOGSPACE (respectivamente NLOGSPACE), entonces DLOGSPACE = P (respectivamente NLOGSPACE = P).
- Si L es PSPACE-completo con respecto a las reducciones *poly-time*, y L está en P (respectivamente NP), entonces P = PSPACE (respectivamente NP = PSPACE).

Estos resultados se demuestran trivialmente (quedan como ejercicio). El mecanismo habitual para agregar un problema completo a una clase es el que vimos para el caso de la NP-completitud, reduciendo desde un problema completo conocido. La completitud también se puede utilizar para probar la igualdad de dos clases. Dadas dos clases C_1 y C_2, si L es C_1-completo y C_2-completo con respecto a las reducciones *poly-time* (respectivamente *log-space*), y C_1 y C_2 son cerradas con respecto a las reducciones *poly-time* (respectivamente *log-space*), entonces $C_1 = C_2$ (la prueba queda como ejercicio).
Enumeramos a continuación ejemplos clásicos de problemas completos de la jerarquía espacio-temporal presentada en la última figura. En primer lugar retomamos el problema de la alcanzabilidad en grafos orientados, y detallamos la demostración de que es NLOGSPACE-difícil con respecto a las reducciones *log-space* para completar la prueba de su completitud iniciada en el Ejemplo 9.2.

Ejemplo 9.3. El problema de la alcanzabilidad en grafos orientados es NLOGSPACE-completo

En el Ejemplo 9.2 probamos que O-ALCANZABILIDAD, el lenguaje que representa el problema de la alcanzabilidad en grafos orientados, pertenece a NLOGSPACE. Para completar la prueba de su completitud, demostramos a continuación que es NLOGSPACE-difícil con respecto a las reducciones *log-space*. Nuevamente recurrimos al método de alcanzabilidad.
Dado $L \in$ NLOGSPACE, sea M una MTN que lo reconoce en espacio logarítmico. Usando que toda configuración de M se puede representar en espacio O(log n), construimos una MTD M_f que trabaja en espacio O(log n) y transforma toda entrada w en una terna (G_w, v_1, v_m), tal que G_w es un grafo orientado con un camino de v_1 a v_m si y

sólo si M acepta w. Los vértices de G_w representan las configuraciones de M a partir de w, salvo el último que es especial. El primer vértice representa la configuración inicial (cabezales apuntando a la primera celda de la cinta respectiva, cinta de trabajo con símbolos blancos, y estado corriente inicial). La MTD M_f trabaja de la siguiente manera:

1. Escribe la configuración inicial de M a partir de w como primer vértice, luego el resto de las configuraciones como siguientes vértices, según el orden canónico y sin repetir el primero, y finalmente un último vértice especial v.
2. Genera una a una, canónicamente, todas las configuraciones C de M, y hace con cada C lo siguiente. Si C no es una configuración final (de aceptación o rechazo), obtiene todas las configuraciones D alcanzables desde C en un paso, y escribe todos los arcos (C, D). Si en cambio C es una configuración final de aceptación, escribe el arco (C, v).
3. Para completar la terna de salida, escribe nuevamente el primer y último vértice de G_w.

Se comprueba fácilmente que G_w tiene un camino del primero al último vértice si y sólo si w está en L(M) (la prueba queda como ejercicio). Además, M_f trabaja en espacio determinístico O(log n) porque mantiene en memoria unas pocas configuraciones de M. La generación de las configuraciones es factible porque la función log n es espacio-construible.

Fin de Ejemplo

Como la alcanzabilidad en grafos orientados (y no orientados) se resuelve en tiempo determinístico polinomial (por ejemplo empleando la técnica de *depth first search*), ahora podemos probar lo que nos faltaba, que NLOGSPACE $\subseteq$ P. Efectivamente, todo lenguaje L de NLOGSPACE cumple L α_{log} O-ALCANZABILIDAD, y por lo tanto también L α_P O-ALCANZABILIDAD. Como O-ALCANZABILIDAD está en P, entonces por ser P cerrada con respecto a las reducciones *poly-time* se cumple que L está en P.

Otro ejemplo de problema NLOGSPACE-completo es 2-SAT (en el Ejemplo 7.3 probamos que está en P). Se puede demostrar mediante los siguientes pasos (que no desarrollamos):

- (Se prueba que) 2-SAT ∈ NLOGSPACE.
- (Hemos probado que) O-ALCANZABILIDAD es NLOGSPACE-completo con respecto a las reducciones *log-space*.
- Como NLOGSPACE es cerrado con respecto al complemento (por el Teorema de Immerman), entonces O-ALCANZABILIDAD^C también es NLOGSPACE-completo con respecto a las reducciones *log-space* (la prueba es trivial y queda como ejercicio).
- (Se prueba que) O-ALCANZABILIDAD^C α_{log} 2-SAT (por lo tanto se cumple que 2-SAT es NLOGSPACE-difícil).

Hemos comentado antes que con las reducciones *poly-time* no podemos identificar la subclase de los problemas más difíciles de P, porque todos los problemas de P se reducen polinomialmente entre sí. Lo mismo sucede en DLOGSPACE con respecto a las reducciones *log-space* (la prueba queda como ejercicio). En cambio, con las reducciones *log-space* podemos encontrar problemas P-completos, como se aprecia en el ejemplo presentado a continuación. Un *circuito booleano* es un grafo orientado sin ciclos, tal que:

- Sus vértices, que en este contexto se denominan *compuertas* o *gates*, tienen grado de entrada 0, 1 o 2.
- Cada compuerta v tiene asociado un tipo t(v) del conjunto {true, false, ∨, ∧, ¬, x_1, x_2, …}. Si t(v) está en {true, false, x_1, x_2, …}, el grado de entrada de v es 0. Si t(v) está en {¬}, el grado de entrada de v es 1. Y si t(v) está en {∨, ∧}, el grado de entrada de v es 2.
- Las compuertas con grado de entrada 0 constituyen la entrada del circuito.
- La compuerta con grado de salida 0 constituye la salida del circuito (también se pueden considerar varias compuertas de salida).

De esta manera, se puede definir que un circuito booleano C computa un valor de verdad a partir de una asignación a sus compuertas de entrada, y plantear problemas relacionados. Por ejemplo, si la entrada de C no incluye variables, se define el problema CIRCUIT VALUE (evaluación de circuito), que consiste en determinar si el valor de verdad computado por C es verdadero. CIRCUIT VALUE está en P, claramente

computar los valores de todas las compuertas, desde la entrada hasta la salida, consume tiempo determinístico polinomial con respecto al tamaño de C. Se prueba además que el problema es P-difícil con respecto a las reducciones *log-space*, por lo que CIRCUIT VALUE es P-completo con respecto a este tipo de reducción. Si en cambio la entrada de C incluye variables, se define el problema CIRCUIT SAT (satisfactibilidad de circuito), que consiste en determinar si existe una asignación de valores de verdad a las compuertas de entrada de C que hace que C compute el valor verdadero. En este caso el problema es NP-completo con respecto a las reducciones *log-space* (es el análogo a SAT). Vamos a volver a los circuitos booleanos en la clase siguiente, cuando nos refiramos a la clase NC de los problemas de resolución polilogarítmica en tiempo paralelo.

Otro ejemplo de problema P-completo es la programación lineal. Se indicó antes que está en P, y se prueba que existe una reducción *log-space* de CIRCUIT VALUE a dicho problema.

El problema QBF (por *quantified boolean formulas*, o fórmulas booleanas cuantificadas) consiste en determinar si una fórmula booleana con cuantificadores y sin variables libres es verdadera. Se prueba que es PSPACE-completo con respecto a las reducciones *poly-time*. El lenguaje que representa el problema es QBF = {φ | φ es una fórmula booleana con cuantificadores, no tiene variables libres, y es verdadera}. Notar que una fórmula booleana φ sin cuantificadores, y con variables x_1, …, x_k, es satisfactible si y sólo si la fórmula booleana $\exists x_1 \exists x_2 \exists x_3 \ldots \exists x_k$ (φ) es verdadera, por lo que QBF generaliza SAT (de hecho también se lo denomina QSAT), y así es NP-difícil con respecto a las reducciones *poly-time*. A QBF no se le conoce resolución que tarde tiempo no determinístico polinomial. Por ejemplo, dada la fórmula $\varphi = \forall x \forall y \forall z\ (x \wedge y \wedge z)$, no alcanza con chequear una asignación de valores de verdad, sino que hay que considerar las 2^3 posibilidades. La prueba de que QBF ∈ PSPACE se basa en la construcción de una función recursiva Eval que trabaja de la siguiente manera (para simplificar la notación, usamos 1 en lugar del valor verdadero y 0 en lugar del valor falso):

1. Eval(1) = 1 y Eval(0) = 0
2. Eval(φ, ¬) = ¬Eval(φ)
3. Eval(φ_1, φ_2, ∧) = Eval(φ_1) ∧ Eval(φ_2) (el ∨ se define de manera similar)
4. Eval(φ, ∀x) = Eval(φ[x | 1]) ∧ Eval(φ[x | 0])

5. $Eval(\varphi, \exists x) = Eval(\varphi[x \mid 1]) \vee Eval(\varphi[x \mid 0])$

donde $\varphi[x \mid i]$ denota la sustitución de la variable x por el valor i en la fórmula φ.
El análisis sintáctico de φ se puede efectuar claramente en espacio determinístico polinomial. Por otro lado, la cantidad de cuantificadores más la cantidad de conectivos de φ es a lo sumo $|\varphi| = n$, y así la profundidad de la recursión y el espacio ocupado por los parámetros de una invocación miden O(n). Por lo tanto, Eval consume espacio determinístico $O(n^2)$. Se demuestra además que QBF es PSPACE-difícil con respecto a las reducciones *poly-time*. No desarrollamos la prueba, la cual es otro ejemplo de reducción en que se relacionan computaciones de MT con fórmulas booleanas.
Distintos problemas que genéricamente se conocen como juegos entre dos personas *(two-person games)*, se prueba que son PSPACE-completos con respecto a las reducciones *poly-time*. Dichos problemas se especifican de la siguiente manera:

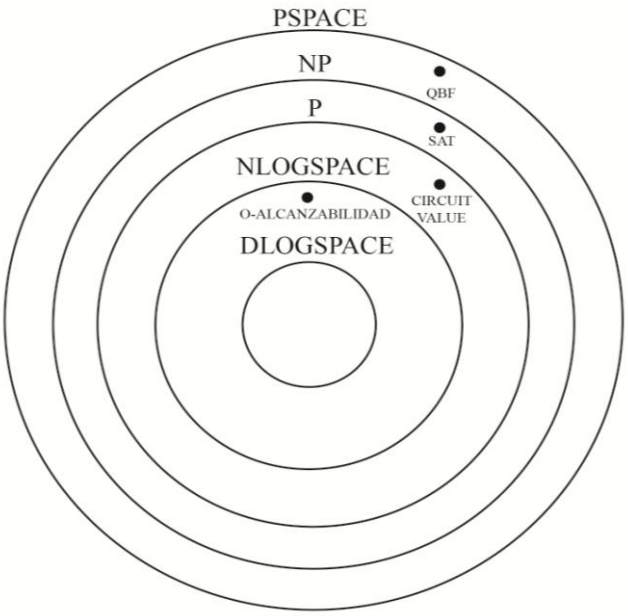
- Dos jugadores, el blanco y el negro, alternan sus jugadas, uno tras otro, impactando sobre un tablero de n x n (u otra estructura, como por ejemplo un grafo). La primera jugada es del blanco.
- Existe una configuración inicial del tablero, y configuraciones finales que se consideran ganadoras para uno u otro.
- El problema consiste en determinar si el blanco gana en k = p(n) jugadas, siendo p(n) un polinomio. En otras palabras, se debe establecer 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 la configuración C_1 a la configuración C_2, existe una jugada del blanco de la configuración C_2 a la configuración C_3, y así sucesivamente hasta llegar a si existe una jugada del blanco de la configuración C_{k-2} a la configuración C_{k-1}, tal que para todas las jugadas del negro de la configuración C_{k-1} a la configuración C_k, la configuración C_k es ganadora para el blanco, con $k = O(n^c)$ para alguna constante c.

En esta gama de juegos se incluye el juego de geografía. En este caso, el blanco elige una ciudad, luego el negro elige otra ciudad cuyo nombre empieza con la letra con la que termina el nombre de la ciudad anterior (no se pueden repetir ciudades), luego sigue el blanco de la misma manera, y así sucesivamente hasta que un jugador no tiene más ciudades para elegir y pierde. El problema se puede formular en términos de un grafo

orientado, tal que sus vértices representan ciudades y existe un arco del vértice v_1 al vértice v_2 si la última letra del nombre de la ciudad de v_1 coincide con la primera letra del nombre de la ciudad de v_2. Se prueba que el problema se resuelve en espacio determinístico polinomial, y que existe una reducción *poly-time* de QBF al mismo.

El mismo problema QBF puede ser visto como un juego. Asumiendo sin perder generalidad que las fórmulas φ de QBF tienen la forma $\exists x_1 \forall x_2 \exists x_3 \ldots Qx_k\ (\varphi)$, con Q = ∃ o ∀ según k sea impar o par, respectivamente, se puede tomar a ∃ y ∀ como dos jugadores. El jugador ∃ mueve primero. La jugada i consiste en asignar un valor de verdad a la variable x_i de la fórmula φ (si k es impar la hace ∃, y si k es par la hace ∀). El jugador ∃ intenta que la fórmula φ resulte verdadera, mientras que su contrincante ∀ intenta que resulte falsa. Obviamente, después de k jugadas alguno de los dos gana. Otros ejemplos clásicos de juegos PSPACE-completos los constituyen el juego de las damas, el go y el hexágono. Para su estudio, los juegos se generalizan a tableros de n x n, y el número de jugadas se acota con un polinomio p(n) (este esquema no resulta naturalmente aplicable al ajedrez, en el que las reglas de terminación producen partidas exponencialmente prolongadas, y el tamaño del tablero y las distintas categorías de las piezas constituyen una parte esencial de la definición del juego).

La figura siguiente es una "fotografía ampliada" del solapamiento de las jerarquías temporal y espacial graficado previamente, ahora incorporando algunos de los problemas completos mencionados:



Ya dicho antes, para encontrar los lenguajes más difíciles de la clase DLOGSPACE se debe recurrir a un tipo de reducción distinta de las que hemos considerado.

Ejercicios de la Clase 9

1. Sea el problema de determinar, dados dos grafos G_1 y G_2, si G_1 tiene un circuito de Hamilton y G_2 no. Indicar si el problema pertenece a CO-NP.
2. Probar que si un problema es NP-completo, entonces su complemento es CO-NP-completo.
3. En la Clase 9 se estableció que si L es un lenguaje NP-completo y está en CO-NP, entonces NP = CO-NP, y se probó el caso NP $\subseteq$ CO-NP. Probar que CO-NP $\subseteq$ NP.
4. Probar que si A es un lenguaje espejo (ver definición en el Ejercicio 6 de la Clase 4) y está en NPC, entonces NP = CO-NP.
5. Una MT no determinística fuerte es una MT no determinística tal que cada una de sus computaciones puede terminar en el estado de aceptación, de rechazo o indefinido (no se sabe la respuesta). Una MT M de este tipo reconoce un lenguaje L si cumple lo siguiente: si w $\in$ L, entonces todas sus computaciones terminan en el estado de aceptación o indefinido, y al menos una lo hace en el estado de aceptación; y si w $\notin$ L, entonces todas sus computaciones terminan en el estado de rechazo o indefinido, y al menos una lo hace en el estado de rechazo. Dada una MT no determinística fuerte M que trabaja en tiempo polinomial, probar que L = L(M) si y sólo si L $\in$ NP $\cap$ CO-NP.
6. Probar (basándose en los conceptos de certificado suscinto y descalificación suscinta) que la factorización, en su forma de problema de decisión, está en NP $\cap$ CO-NP.
7. Sea f una función de los enteros de longitud k a los enteros de longitud k, computable en tiempo polinomial y tal que f^{-1} no es computable en tiempo polinomial. Probar que el lenguaje de pares $\{(x, y) \mid f^{-1}(x) < y\}$ pertenece a la clase (NP $\cap$ CO-NP) – P.
8. Justificar:
   i. Si una función espacial S cumple para toda cadena w que $S(|w|) \geq |w|$, entonces no hace falta recurrir al modelo de MT con una cinta de entrada de sólo lectura.

ii. Las MT con una cinta de entrada de sólo lectura pueden ser también las MT estándar en el marco de la complejidad temporal.

9. Probar que si $S_1(n) = O(S_2(n))$, entonces $DSPACE(S_1(n)) \subseteq DSPACE(S_2(n))$.
10. Dada una función S(n) espacio-construible, construir una MT que a partir de una cadena w genere una cadena de S(n) símbolos X.
11. Probar que $DSPACE(S(n)) \subseteq R$:
    i. Asumiendo que $S(n) \geq \log_2 n$ es espacio-construible.
    ii. Sin la asunción anterior.
    iii. Lo mismo que i y ii pero para el caso de NSPACE(S(n)).
12. Probar que al igual que la jerarquía temporal, la jerarquía espacial es densa: si S(n) es una función total computable entonces existe un lenguaje recursivo L tal que $L \notin DSPACE(S(n))$.
13. Completar la prueba del Ejemplo 9.1.
14. Completar la prueba del Ejemplo 9.2.
15. Probar:
    i. $DSPACE(\log^k(n)) \subset DSPACE(\log^{k+1} n)$, para todo $k \geq 1$.
    ii. NLOGSPACE $\subset$ PSPACE.
    iii. NP $\cup$ CO-NP $\subseteq$ PSPACE.
16. Sea FPSPACE el conjunto de las funciones totales $f: \Sigma^* \longrightarrow N$ computables en espacio determinístico polinomial, y #FP el conjunto de las funciones totales $g: \Sigma^* \longrightarrow N$ tales que $g \in$ #FP si y sólo si existe una MTN M que trabaja en tiempo polinomial y que a toda entrada w la acepta en exactamente g(w) computaciones. Probar que #FP $\subseteq$ FPSPACE.
17. Probar:
    i. Toda reducción *log-space* es una reducción *poly-time*.
    ii. Como en el caso de las reducciones *poly-time* con respecto a P, en que todos los problemas de P se reducen polinomialmente entre sí, lo mismo sucede en DLOGSPACE con respecto a las reducciones *log-space*.
18. Probar:
    i. Si L es NLOGSPACE-completo con respecto a las reducciones *log-space*, y L está en DLOGSPACE, entonces DLOGSPACE = NLOGSPACE.
    ii. Si L es P-completo con respecto a las reducciones *log-space*, y L está en DLOGSPACE (respectivamente NLOGSPACE), entonces DLOGSPACE = P (respectivamente NLOGSPACE = P).

iii. Si L es PSPACE-completo con respecto a las reducciones *poly-time*, y L está en P (respectivamente NP), entonces P = PSPACE (respectivamente NP = PSPACE).

iv. Dadas dos clases C_1 y C_2, si L es C_1-completo y C_2-completo con respecto a las reducciones *poly-time* (respectivamente *log-space*), y C_1 y C_2 son cerradas con respecto a las reducciones *poly-time* (respectivamente *log-space*), entonces $C_1 = C_2$.

19. Completar la prueba del Ejemplo 9.3.
20. Probar que como NLOGSPACE es cerrado con respecto al complemento por el Teorema de Immerman, entonces O-ALCANZABILIDAD^C también es NLOGSPACE-completo con respecto a las reducciones *log-space*.
21. Probar que si P = PSPACE, entonces toda función computable en espacio polinomial pertenece a FP.
22. Probar que 2-SAT es NLOGSPACE-completo.
23. Probar que el juego de geografía es PSPACE-completo.

Clase 10. Misceláneas de complejidad computacional

TEMA 10.1. PROBLEMAS DE BÚSQUEDA

Hemos trabajado hasta ahora con el subconjunto de los problemas de decisión. Los problemas más generales son los de *búsqueda*, el tema de esta sección, los cuales se resuelven por MT que no sólo aceptan cuando una instancia tiene solución, sino que además generan una. Por ejemplo, en el caso del problema de búsqueda asociado a SAT, la resolución consiste en generar una asignación de valores de verdad A que satisface una fórmula booleana φ, o responder que no hay solución si φ no es satisfactible.

Analizar la complejidad computacional temporal en términos de los problemas de decisión nos facilitó la presentación de los temas, y sin perder en esencia generalidad al focalizarnos en la cuestión de su resolución eficiente o ineficiente. Es que de la imposibilidad de determinar eficientemente la existencia de una solución se puede inferir la imposibilidad de encontrar eficientemente alguna. Por ejemplo, asumiendo $P \neq NP$, encontrar una asignación A que satisface una fórmula φ no puede tardar tiempo determinístico polinomial, porque SAT es NP-completo: si existe un algoritmo que eficientemente encuentra A o responde que no hay solución, entonces SAT se puede resolver también eficientemente invocando a dicho algoritmo (absurdo si $P \neq NP$). El mismo razonamiento se puede aplicar sobre una tercera clase de problemas, los problemas de *enumeración*, que consisten en calcular la cantidad de soluciones de las instancias.

Se identifica con FNP a la clase de los problemas de búsqueda tales que los problemas de decisión asociados están en NP. La F de FNP se debe a que a estos problemas también se los conoce como *problemas de función* (*function problems*). Si bien una instancia puede tener varias soluciones, se acostumbra a emplear en la literatura esta denominación. Análogamente, FP ahora no sólo será la clase de las funciones computables en tiempo determinístico polinomial, sino que también incluirá a los problemas de búsqueda tales que los problemas de decisión asociados están en P. Vamos a considerar solamente problemas de búsqueda con soluciones de tamaño polinomial con respecto al de sus instancias, porque de lo contrario no podrían resolverse eficientemente.

Una manera habitual de analizar conjuntamente los problemas de búsqueda y decisión se basa en el uso de las *Cook-reducciones*. Un problema de búsqueda o de decisión A es Cook-reducible a un problema de búsqueda o de decisión B, si existe una MTD M con oráculo B, es decir M^B, que resuelve A en tiempo polinomial. Por lo tanto, una Cook-reducción se asemeja a una Turing-reducción (definida en el Tema 5.4), pero se distingue por lo siguiente:

- A y B pueden ser problemas de búsqueda o de decisión (si B es un problema de búsqueda, el oráculo devuelve en un paso una solución de B).
- La MTD M^B trabaja en tiempo polinomial.

De esta manera, las reducciones *poly-time* o Karp-reducciones constituyen un caso particular de las Cook-reducciones, en las que los dos problemas son de decisión y el oráculo B se puede invocar sólo una vez y al final. Otro caso particular es el de las reducciones entre problemas de búsqueda denominadas *Levin-reducciones*. Una Levin-reducción de A a B consiste en dos funciones f y g de FP, tales que si w es una instancia de A, entonces f(w) es una instancia de B, y si z es una solución de f(w), entonces g(z) es una solución de w. Considerando este último tipo de reducción, se define que un problema de búsqueda A es *FNP-completo* si está en FNP y todos los problemas de FNP son Levin-reducibles a él.

Como siempre, que exista una reducción de A a B, en este caso una Cook-reducción, significa que B es tan o más difícil que A. Por ejemplo, si FSAT es el problema de búsqueda asociado a SAT, que SAT sea Cook-reducible a FSAT significa que FSAT no puede estar en FP, a menos que P = NP. El siguiente es un primer ejemplo de aplicación de las Cook-reducciones, y trata justamente la relación entre los problemas SAT y FSAT.

Ejemplo 10.1. Cook-reducciones entre los problemas SAT y FSAT

Se prueba trivialmente que SAT es Cook-reducible a FSAT. La siguiente MTD M^{FSAT} resuelve SAT eficientemente a partir de una fórmula booleana φ válida sintácticamente: invoca a FSAT con φ, y acepta si y sólo si el oráculo devuelve una asignación.

Menos intuitivo es el caso inverso, pero también se cumple que FSAT es Cook-reducible a SAT. La siguiente MTD M^{SAT}, a partir de una fórmula booleana φ válida sintácticamente, con variables $x_1, \ldots, x_m$, resuelve FSAT eficientemente:

1. Invoca a SAT con φ. Si el oráculo responde negativamente, entonces responde que no hay solución.
2. Simplifica φ haciendo x_1 := verdadero, e invoca a SAT con la fórmula reducida. Si el oráculo responde negativamente, retoma la forma original de φ y la simplifica haciendo ahora x_1 := falso (como φ es satisfactible, si no lo es con x_1 con valor verdadero entonces lo es con x_1 con valor falso).
3. Repite el paso 2 considerando la variable x_2, luego la variable x_3, y así sucesivamente hasta llegar a la variable x_m, encontrando de esta manera una asignación A que satisface φ.

Fin de Ejemplo

Así, SAT es Cook-reducible a FSAT y FSAT es Cook-reducible a SAT. Se dice en este caso que los problemas son *polinomialmente equivalentes* (comparar con la definición de problemas recursivamente equivalentes que se formula en el Tema 5.4). Efectivamente, acabamos de probar que SAT y FSAT son igualmente difíciles. También se define que FSAT es *auto-reducible*: es Cook-reducible al problema de decisión asociado (hay otra acepción de auto-reducibilidad, restringida a las Cook-reducciones de los problemas de decisión a sí mismos, de modo tal que las invocaciones al oráculo deben hacerse con cadenas más pequeñas que las instancias; por ejemplo, SAT es auto-reducible considerando esta definición alternativa). La auto-reducibilidad es una propiedad de muchos problemas de búsqueda naturales, entre ellos todos los problemas FNP-completos, lo que confirma la relevancia del estudio de los problemas de decisión. Un ejemplo de problema de búsqueda auto-reducible no FNP-completo sería el de los grafos isomorfos.

El siguiente es otro ejemplo de aplicación de las Cook-reducciones, ahora referido a un problema de optimización, FTSP, la versión del problema del viajante de comercio en que hay que encontrar el recorrido mínimo.

Ejemplo 10.2. Cook-reducciones entre los problemas TSP y FTSP

En términos de un grafo G con arcos con costos, el problema FTSP consiste en encontrar el circuito de Hamilton de costo mínimo de G. Recordar que el lenguaje que representa el problema de decisión asociado es TSP = {(G, B) | G tiene un circuito de Hamilton con costo menor o igual que B}. Se prueba trivialmente que TSP es Cook-

reducible a FTSP (queda como ejercicio). Probamos a continuación que también existe una Cook-reducción de FTSP a TSP. Vamos a construir una MTD M^{TSP} que resuelve FTSP en tiempo polinomial. A partir de un grafo válido G, M^{TSP} hace:

1. Primero obtiene el costo C del circuito de Hamilton de costo mínimo, invocando a TSP varias veces sin modificar el parámetro G, pero sí cambiando cada vez el parámetro B. Como C varía entre 0 y 2^n, siendo n = |G|, entonces por búsqueda binaria C se puede calcular luego de O(n) invocaciones.
2. Luego obtiene el circuito de Hamilton de costo mínimo, invocando a TSP varias veces sin modificar el parámetro B (con valor C), pero sí cambiando cada vez el grafo G. En toda invocación modifica el costo de un arco distinto, asignándole el valor C + 1. Que el oráculo acepte o rechace significa que dicho arco no forma parte o sí lo hace del circuito de Hamilton mínimo, respectivamente. En el segundo caso marca el arco y le restituye su costo original. Así, al cabo de |E| iteraciones obtiene la solución.

Fin de Ejemplo

De esta manera, FTSP y TSP son polinomialmente equivalentes. En realidad, esta equivalencia se generaliza a todos los problemas de búsqueda de un óptimo (máximo o mínimo) con respecto a los problemas asociados de búsqueda o de decisión con una cota o *umbral* (*threshold*), lo que no es del todo intuitivo, porque los problemas de búsqueda de un óptimo no necesariamente están en FNP (verificar una solución óptima no es lo mismo que verificar una solución cualquiera).

Otra técnica clásica para el estudio de la complejidad temporal de los problemas de búsqueda la constituyen las *aproximaciones polinomiales*, justamente ante la imposibilidad de encontrar óptimos eficientemente (sustentada por la NP-completitud de los problemas de decisión asociados). La idea es construir un algoritmo eficiente que produzca una solución "buena", cercana al óptimo según un criterio determinado. En el ejemplo siguiente se presenta una aproximación polinomial para FVC, el problema de búsqueda del cubrimiento de vértices mínimo de un grafo.

Ejemplo 10.3. Aproximación polinomial para el problema FVC

El problema de decisión del cubrimiento de vértices de un grafo es NP-completo (ver Ejemplo 8.5), y entonces, a menos que P = NP, no existe un algoritmo eficiente para

encontrar cubrimientos mínimos. Vamos a construir una aproximación polinomial para este problema. Dado un grafo válido G = (V, E), la MT siguiente genera en tiempo determinístico polinomial un cubrimiento de vértices V' $\subseteq$ V de G (luego indicamos cuán cerca está del óptimo):

1. Hace V' := $\emptyset$ y E' := E.
2. Si E' = $\emptyset$, acepta.
3. Hace E' := E' – {(v_1, v_2)}, siendo (v_1, v_2) algún arco de E'.
4. Si $v_1 \notin$ V' y $v_2 \notin$ V', entonces hace V' := V' $\cup$ {v_1, v_2}.
5. Vuelve al paso 2.

V' cubre un conjunto A de arcos no adyacentes de G. Se cumple que |A| = |V'| / 2 porque sólo los dos vértices de cada uno de estos arcos están en V'. Por otro lado, cualquier cubrimiento de vértices C de G debe incluir un vértice de todo arco de A, por lo que debe medir al menos |C| = |V'| / 2. De este modo, el tamaño de V' es a lo sumo el doble del óptimo. Claramente, el algoritmo trabaja en tiempo determinístico polinomial (itera |E| veces).

Fin de Ejemplo

Dado un problema de búsqueda, si llamamos opt(w) a la solución óptima de una instancia w, y m(M(w)) a la medida de la solución M(w) obtenida por una aproximación polinomial M para el problema, se define el *error relativo* ε de M de la siguiente manera. Para todo w:

$$\varepsilon = |m(M(w)) - opt(w)| / \max(m(M(w)), opt(w))$$

Notar que el valor de ε varía entre 0 y 1. En el ejemplo anterior, entonces, se cumple que $\varepsilon \leq 1/2$ (se dice que el algoritmo construido es una 1/2-aproximación polinomial). Obviamente, el objetivo es encontrar aproximaciones polinomiales con un error relativo lo más pequeño posible, que se conoce como *umbral de aproximación*. Si el umbral de aproximación está cerca del 0 significa que la solución se puede aproximar al óptimo arbitrariamente, y si está cerca del 1, que no hay aproximación polinomial posible. Se prueba que si P $\neq$ NP, entonces existen problemas no aproximables.

Por ejemplo, se demuestra que si existe una Ɛ-aproximación polinomial para el problema de optimización del viajante de comercio, con $\varepsilon < 1$, entonces P = NP. En el otro extremo se encuentra el problema de la mochila, en el que se plantea un conjunto de m items, cada uno calificado por dos valores, por ejemplo su volumen v_i y su peso p_i, y se pretende encontrar un subconjunto de items que maximice la suma de los v_i y que cumpla que la suma de los p_i asociados no supere una cota determinada. En este caso se prueba que existe una Ɛ-aproximación para todo $\varepsilon > 0$ (cabe remarcar que el problema de decisión de la mochila es NP-completo). El comportamiento de los umbrales de aproximación es muy variable, y una razón es que las reducciones de problemas que se utilizan en este marco no necesariamente preservan las características de los mismos.

Los problemas aproximables conforman la clase APX. En particular, la clase PAS (por *polynomial approximation scheme* o esquema de aproximación polinomial), incluida en APX, nuclea a los problemas Ɛ-aproximables para todo Ɛ. Como es habitual, se emplean reducciones de problemas y el concepto de completitud para el poblamiento de dichas clases y para relacionarlas con el resto. Una de las reducciones consideradas es la *APX-reducción*, que preserva la propiedad de ser aproximable. Asumiendo $P \neq NP$, como APX es cerrada con respecto a las APX-reducciones, entonces la completitud con respecto a dichas reducciones permite encontrar problemas no aproximables. Se conocen pocos problemas completos de esta naturaleza, en comparación con el tamaño de la clase NPC. Las pruebas son más dificultosas, y además muchos problemas aproximables se asocian a problemas de decisión NP-completos.

TEMA 10.2. OTROS USOS DE LOS ORÁCULOS

Los enunciados referidos a los pares de clases P y NP, NP y CO-NP, EXP y NEXP, etc., se pueden *relativizar* por medio de los oráculos. Por ejemplo, si P^L y NP^L son las clases de los lenguajes reconocibles por MTD y MTN, respectivamente, con oráculo L, que trabajan en tiempo polinomial, entonces $P^L \neq NP^L$ es una conjetura relativizada. Conjeturas como ésta pueden servir para probar conjeturas no relativizadas. Siguiendo con el ejemplo, si valiese $P^L \neq NP^L$, entonces un camino para probar $P \neq NP$ podría consistir en refinar en varios pasos el oráculo L hasta llegar a uno tan trivial que nos permita formular la conjetura no relativizada. Se han probado enunciados en esta dirección, como los siguientes:

- Para todo oráculo L se cumple P = NP si y sólo si $P^L = NP_b^L$, tal que NP_b^L es la subclase de NP^L de los lenguajes reconocibles por MT que invocan a L un número polinomial de veces, considerando todas las computaciones.
- Para todo lenguaje unitario U se cumple P = NP si y sólo $P^U = NP^U$, donde un lenguaje es unitario si se basa en un alfabeto de un solo símbolo.

La relativización permite además entender qué técnicas pueden servir para encarar las pruebas de las conjeturas. Por ejemplo, se prueba que existen oráculos A y B tales que $P^A = NP^A$ y $P^B \neq NP^B$ (Teorema de Baker, Gill y Solovay). Por lo tanto, toda técnica a emplear para encontrar la relación entre P y NP debe ser no relativizable, es decir, debe tener la propiedad de que lo que pruebe sin oráculos no necesariamente deba valer con oráculos. Más precisamente, si se probara por diagonalización que P ≠ NP, la misma demostración aplicaría utilizando oráculos, pero como existe un oráculo A tal que $P^A = NP^A$, entonces concluimos que la diagonalización no es un camino válido para demostrar la conjetura. Lo mismo ocurre con la técnica de simulación pero ahora para probar P = NP, por existir un oráculo B tal que $P^B \neq NP^B$. En cambio, una aproximación no relativizable podría ser encontrar alguna propiedad algebraica en P que no se cumpla en NP. Se presenta a continuación un ejemplo sencillo de relativización (parte del teorema recién mencionado).

Ejemplo 10.4. Prueba de la conjetura relativizada $P^{QBF} = NP^{QBF}$

Dijimos en la clase pasada que QBF (el lenguaje de las fórmulas booleanas, con cuantificadores y sin variables libres, que son verdaderas) es PSPACE-completo, y por lo tanto más difícil que P y NP asumiendo NP ≠ PSPACE. Intuitivamente, es razonable que P y NP, valiéndose de QBF como oráculo, resulten iguales. Vamos a probar $P^{QBF} = NP^{QBF}$. Se cumple $P^{QBF} \subseteq NP^{QBF}$ por definición. Para probar $NP^{QBF} \subseteq P^{QBF}$ se demostrará primero $NP^{QBF} \subseteq$ PSPACE y luego PSPACE $\subseteq P^{QBF}$.

- $NP^{QBF} \subseteq$ PSPACE. Sea $L \in NP^{QBF}$ y M_1^{QBF} una MTN que reconoce L en tiempo polinomial. Dada una entrada w, M_1^{QBF} efectúa en cada computación un número polinomial de invocaciones a QBF. Toda vez, el tamaño de la pregunta es polinomial con respecto al tamaño de w. La siguiente MTD M_2 reconoce L en espacio polinomial: simula M_1^{QBF} computación por computación reutilizando

espacio, toda invocación a QBF la reemplaza ejecutando una MTD que reconoce QBF en espacio polinomial, y acepta si y sólo si alguna computación acepta. Queda como ejercicio probar que $L(M_2) = L$ y que M_2 trabaja en espacio determinísitico polinomial.

- PSPACE $\subseteq P^{QBF}$. Sea $L \in$ PSPACE y M_f una MTD que computa una reducción polinomial de L a QBF (posible porque QBF es PSPACE-completo). La siguiente MTD M_3^{QBF} reconoce L en tiempo polinomial: dada una entrada w, ejecuta M_f para calcular f(w), invoca al oráculo QBF con f(w), y acepta si y sólo sí el oráculo acepta. Queda como ejercicio probar que $L(M_3^{QBF}) = L$ y que M_3^{QBF} trabaja en tiempo determinísitico polinomial.

Fin de Ejemplo

Notar que la relativización anterior vale para cualquier lenguaje PSPACE-completo, y entonces se puede formular que $P^L = NP^L$ para todo oráculo L que sea PSPACE-completo.

Por medio de los oráculos también se puede definir otra jerarquía de complejidad, útil para clasificar importantes problemas. Dicha jerarquía se solapa con la jerarquía espacio-temporal presentada en la clase anterior, cubriendo la franja de P a PSPACE, y se denomina *jerarquía polinomial*. Por cómo se formula, se la considera la jerarquía análoga a la *jerarquía aritmética* o *jerarquía de Kleene* de la computabilidad, en la que en cada nivel se definen problemas indecidibles más difíciles que en el nivel anterior. Sea P^{NP} la clase de todos los lenguajes reconocidos en tiempo determinístico polinomial por MT con un oráculo de NP (en realidad es lo mismo si se fija un único oráculo NP-completo, por ejemplo SAT). De la misma manera se definen NP^{NP} y CO-NP^{NP}. La jerarquía polinomial es el siguiente conjunto infinito de clases de lenguajes $\Delta_k P$, $\Sigma_k P$ y $\Pi_k P$:

- En el nivel 0 se definen $\Delta_0 P$, $\Sigma_0 P$ y $\Pi_0 P$, los tres iguales a P
- En los niveles 1, 2, …, se definen, para $k \geq 0$:

  $\Delta_{k+1} P = P^{\Sigma k P}$

  $\Sigma_{k+1} P = NP^{\Sigma k P}$

  $\Pi_{k+1} P = \text{CO-}NP^{\Sigma k P}$

Así, la jerarquía polinomial tiene a la clase P en el nivel 0; a las clases P, NP y CO-NP en el nivel 1 (simplificando); a las clases P^{NP}, NP^{NP} y $CO\text{-}NP^{NP}$ en el nivel 2; etc. Es natural asumir que las tres clases del nivel 1 en adelante son distintas y se relacionan de la misma manera. Toda clase de un nivel incluye a todas las clases de los niveles inferiores. De esta manera, alcanza con definir a la jerarquía polinomial como la unión infinita de las clases $\Sigma_k P$. Se la denota con PH (por *polynomial hierarchy* o jerarquía polinomial). Se prueba que PH $\subseteq$ PSPACE. Por otra parte, no se sabe si a partir de un determinado k se cumple $\Sigma_k P = \Sigma_{k+1} P$, lo que se conoce como *colapso* de la jerarquía polinomial. La jerarquía podría colapsar aún si P $\neq$ NP. Un ejemplo de problema clasificado en términos de la jerarquía polinomial es el del circuito booleano mínimo. El problema consiste en establecer si un circuito booleano es el más pequeño que existe para computar una función booleana determinada. Se prueba que el problema pertenece a la clase $\Pi_2 P = CO\text{-}NP^{NP}$, y no se sabe si es completo en la clase. Otro problema clásico de la jerarquía polinomial es el problema QBF_k, caso particular de QBF con fórmulas de la forma

$$\exists X_1 \forall X_2 \exists X_3 \ldots Q X_k \, (\varphi)$$

siendo Q = $\exists$ o $\forall$ según k sea impar o par, respectivamente, y $X_1, \ldots, X_k$ una partición de las variables de la fórmula φ. Se prueba que QBF_k es $\Sigma_k P$-completo. Se han encontrado pocos problemas completos en la jerarquía polinomial. En particular, no se conocen problemas PH-completos. Probablemente no existan, porque se prueba que si se encuentra un problema PH-completo, entonces la jerarquía colapsa. Por lo tanto, PH = PSPACE es otra condición suficiente para que la jerarquía polinomial colapse, ya que PSPACE tiene problemas completos. También en este caso existe una caracterización provista por la complejidad descriptiva. Se demuestra que la clase PH coincide con los problemas que pueden especificarse mediante la lógica de segundo orden (ahora completa, es decir no restringida a la lógica existencial, que es la que corresponde a NP, ni a la lógica universal, que es la que corresponde a CO-NP).

TEMA 10.3. MÁQUINAS DE TURING PROBABILÍSTICAS

Los *algoritmos probabilísticos* conforman una alternativa muy útil para el estudio de la complejidad temporal de los problemas. Un algoritmo de este tipo es de tiempo

polinomial, y en determinados pasos elige aleatoriamente una continuación entre varias, lo que puede provocar que genere diferentes salidas en diferentes ejecuciones. Conociendo la distribución de las salidas, se efectúan asunciones con una determinada probabilidad.

El modelo computacional asociado lo constituyen las *máquinas de Turing probabilísticas*, las cuales se definen como MT no determinísticas que trabajan en tiempo polinomial y tienen un criterio de aceptación distinto al considerado hasta ahora: aceptan o rechazan una entrada de acuerdo a la relación que se cumple entre la cantidad de computaciones de aceptación y rechazo. Sin perder generalidad, se trabaja con MTN con grado no determinístico dos, todos los pasos son no determinísticos, y todas las computaciones ejecutan la misma cantidad polinomial de pasos con respecto a la longitud de la entrada.

Esta aproximación, que es la que tendremos en cuenta, se denomina *on-line* (en línea, y también se la conoce como de *tiro de moneda* o *coin-flipping*). Una aproximación alternativa, denominada *off-line* (fuera de línea), consiste en manejar la aleatoriedad por medio de una segunda cadena de entrada, que es una secuencia de unos y ceros cuya longitud es la cantidad polinomial de pasos a ejecutar, determinando así para cada paso la elección a adoptar.

Se definen distintos tipos de MT probabilísticas, y correspondientemente clases de problemas con el mismo nombre, según el criterio de aceptación que se adopte. Por ejemplo, una *máquina PP* (por *probabilistic polynomial time*, o tiempo polinomial probabilístico) acepta una entrada si y sólo si la acepta en más de la mitad de sus computaciones. Siendo PP la clase de problemas asociada a las máquinas PP, se demuestra fácilmente que NP $\subseteq$ PP (la prueba queda como ejercicio).

Un segundo tipo de MT probabilística es la *máquina RP* (por *randomized polynomial time*, o tiempo polinomial aleatorio). Se define de la siguiente manera:

- Dada una entrada, la acepta en al menos 1/2 de sus computaciones o la rechaza en todas.
- Acepta una entrada si y sólo si la acepta en al menos 1/2 de sus computaciones.

En el ejemplo que se presenta a continuación, se describe una máquina RP para el problema de composicionalidad.

Ejemplo 10.5. MT probabilística para la composicionalidad

Vamos a construir una máquina RP M para establecer, con una determinada probabilidad de error, si un número natural m es compuesto (no es primo). El algoritmo se basa en una condición de composicionalidad derivada de un teorema de Fermat: si m es un número natural compuesto impar, entonces tiene al menos (m – 1) / 2 *testigos de composicionalidad*, siendo x un testigo de composicionalidad de m si $1 \leq x < m$ y $x^{m-1} \neq 1 \pmod{m}$, o bien existen números naturales z, i que cumplen $z = (m - 1) / 2^i$, y $x^z - 1$ y m tienen un divisor común diferente de 1 y m. Dado un número natural m, la máquina RP M hace (para simplificar, no se considera que el grado del no determinismo de M sea dos, ni que todos sus pasos sean no determinísticos):

1. Si m es par, acepta.
2. Hace x := random(1, m – 1).
3. Si x es un testigo de composicionalidad, acepta, y en caso contrario, rechaza.

La función random(1, m – 1) selecciona aleatoriamente un número entre 1 y m – 1. Se prueba que se puede determinar eficientemente si x es un testigo de composicionalidad. Notar que si m es primo, todas las computaciones de M rechazan, y así la probabilidad de error es cero, y si m es compuesto, al menos la mitad de las computaciones de M aceptan, por lo que la probabilidad de error en este caso es a lo sumo 1/2.

Fin de Ejemplo

Siguiendo con la máquina RP M del ejemplo, la probabilidad de error para determinar si un número es compuesto se puede achicar arbitrariamente de la siguiente manera:

- Se itera k veces la máquina M a partir de un mismo número m, por ejemplo 100 veces.
- Si en alguna iteración M acepta, entonces se acepta: m es un número compuesto porque es par o porque es impar y tiene un testigo de composicionalidad. En este caso la probabilidad de error es cero.
- Si en cambio al cabo de las 100 iteraciones M rechaza, entonces se rechaza: m es un número primo y por eso no se encuentran testigos de composicionalidad, o

bien es un número compuesto. En este último caso, la probabilidad de error es a lo sumo 2^{-100}, asumiendo que toda iteración es independiente de las anteriores.

Este mecanismo de iteración aplica a cualquier problema de la clase RP asociada a las máquinas RP. Las máquinas RP nunca aceptan mal. Pueden rechazar mal, pero con una probabilidad arbitrariamente pequeña si se las itera un número hasta polinomial de veces con respecto a la longitud de las entradas (con las máquinas PP no ocurre lo mismo, para alcanzar una probabilidad de error arbitrariamente pequeña puede llegar a necesitarse iterar un número exponencial de veces). Notar que en base a este esquema iterativo, la clase RP no cambia si se modifica la definición de las máquinas RP, reemplazando la fracción 1/2 por otra menor. Se prueba fácilmente que $P \subseteq RP$ (queda como ejercicio).

Si CO-RP es la clase complemento de RP, entonces las máquinas asociadas ahora pueden aceptar erróneamente (con probabilidad a lo sumo 1/2) y no pueden rechazar erróneamente. Por lo tanto, la clase $RP \cap$ CO-RP incluye lenguajes que se reconocen por medio de dos algoritmos probabilísticos distintos, uno sin aceptaciones erróneas y otro sin rechazos erróneos. Ejecutando los dos algoritmos independientemente k veces, la probabilidad de no obtener una respuesta definitiva es a lo sumo 2^{-k}. La diferencia de este algoritmo compuesto con el que vimos antes es que nunca responde erróneamente. La probabilidad de no tener una respuesta definitiva puede hacerse arbitrariamente pequeña. Existe un algoritmo de este tipo para el problema de primalidad, que utiliza el que mostramos en el ejemplo anterior para la composicionalidad, y que se vale de una propiedad que, al igual que los números compuestos, también tienen los números primos, de abundancia de testigos (en este caso *testigos de primalidad*), chequeable eficientemente. La idea general de dicho algoritmo compuesto es la siguiente. Dado un número natural m:

1. Elige aleatoriamente un posible testigo de primalidad x, y chequea si efectivamente x es un testigo de primalidad. Si lo es, acepta.
2. Elige aleatoriamente un posible testigo de composicionalidad z, y chequea si efectivamente z es un testigo de composicionalidad. Si lo es, rechaza.
3. Si los pasos 1 y 2 se iteraron un número suficiente de veces, entonces responde que no hay respuesta, y en caso contrario vuelve al paso 1.

Los algoritmos de este tipo, en que puede haber además de aceptación y rechazo una respuesta indefinida, se conocen como *algoritmos de Las Vegas* (para diferenciarlos de los otros, que se denominan *algoritmos de Monte Carlo*). Las máquinas probabilísiticas correspondientes se denominan ZPP (por *zero probabilistic polynomial time*, o tiempo polinomial probabilístico con cero probabilidad de error), lo mismo que la clase de problemas asociada, definida entonces como ZPP = RP $\cap$ CO-RP. Claramente, se cumple que P $\subseteq$ ZPP (la prueba queda como ejercicio).

El último tipo de MT probabilística que vamos a presentar es la máquina BPP (por *bounded probabilistic polynomial time*, o tiempo polinomial probabilístico acotado). Se define de la siguiente manera:

- Dada una entrada, la acepta en al menos 3/4 de sus computaciones o la rechaza en al menos 3/4 de sus computaciones.
- Acepta una entrada si y sólo si la acepta en al menos 3/4 de sus computaciones.

Es decir que esta máquina es otra variante de los algoritmos de Monte Carlo. Acepta por clara mayoría o rechaza por clara mayoría. Como en el caso de las máquinas RP, se puede modificar su definición sin alterar la clase de problemas BPP asociada (la fracción 3/4 se puede reemplazar por cualquier otra mayor que 1/2). Y al igual que con las máquinas RP y ZPP, se pueden obtener probabilidades de error muy bajas iterando una cantidad polinomial de veces. Se cumple que RP $\subseteq$ BPP $\subseteq$ PP (la prueba queda como ejercicio).

Integrando las relaciones entre las clases probabilísticas mencionadas anteriormente, queda:

- P $\subseteq$ ZPP $\subseteq$ RP $\subseteq$ BPP $\subseteq$ PP
- P $\subseteq$ ZPP $\subseteq$ RP $\subseteq$ NP $\subseteq$ PP

No se sabe si las inclusiones son estrictas. También se desconoce si BPP $\subseteq$ NP. Es habitual de todos modos considerar tratables a los problemas de ZPP, RP y BPP (bajo la hipótesis de que los algoritmos probabilísticos son implementables, es decir que se cuenta con generadores confiables de números aleatorios), y así no se espera que alguna máquina ZPP, RP o BPP resuelva un problema NP-completo. En la práctica, a veces el

uso de estas máquinas apunta a resolver más simple o eficientemente que las MT determinísticas determinados problemas de P y NPI.

En la complejidad espacial también se definen clases probabilísticas, análogas a las anteriores, como por ejemplo RL y BPL. En este caso, las máquinas probabilísticas trabajan en espacio logarítmico. Al igual que la conjetura P = RP = BPP, se suelen considerar las igualdades DLOGSPACE = RL = BPL.

Otra utilización de la teoría de probabilidades, que ya mencionamos en la Clase 6, es la que se relaciona con el criterio del caso promedio en lugar del peor caso para el cálculo del tiempo de un algoritmo. En este caso se debe considerar la distribución de la población de las entradas de las MT, lo que comúnmente es una tarea difícil. Cabe destacar que existen problemas NP-completos que resultan tratables si se considera el tiempo promedio de resolución. Un área en la que particularmente no aplica el criterio del peor caso es la criptografía de clave pública, basada en la posibilidad de que existan funciones difíciles de invertir. Nos referimos a las funciones *one-way* (o de sentido único). Estas son funciones inyectivas f que se computan eficientemente, tales que para toda cadena w, f(w) es a lo sumo polinomialmente más grande o más pequeña que w, y la función inversa f^{-1} no es computable eficientemente (es decir que las cadenas de salida de la función inversa de una función *one-way* son ineficientemente computables pero eficientemente chequeables). Por lo tanto, sólo si P ≠ NP pueden existir funciones de este tipo. Una función *one-way* candidata es la multiplicación de enteros. Si bien existen algoritmos de tiempo subexponencial, al día de hoy no se conoce una manera eficiente para obtener los factores del producto de dos números primos muy grandes, es decir para factorizarlo. Otra función candidata es la exponenciación módulo primo, denotada con r^x mod p, siendo p un número primo. Invertir esta función, es decir obtener x, es un problema bien conocido, computacionalmente difícil, denominado *logaritmo discreto*. Se han encontrado *algoritmos cuánticos* polinomiales que resuelven este último problema y la factorización. Lo cierto es que la criptografía de clave pública se basa en el supuesto de la existencia de funciones *one-way*. El esquema de clave pública trabaja esencialmente con dos claves, una clave pública c y una clave privada d. La transmisión de un mensaje w se efectúa encriptándolo mediante un algoritmo polinomial C(c, w) = v, mientras que la decodificación del mensaje se lleva a cabo por medio de otro algoritmo D(d, v) = w, de modo tal que no se puede obtener eficientemente la clave d a partir de la clave c, ni el mensaje w a partir del mensaje v sin conocer la clave d. Naturalmente, no es aceptable en este esquema tratar con el criterio

del peor caso (por ejemplo, no es razonable permitir que la mitad de los mensajes encriptados sean fácilmente decodificados). De esta manera se maneja una acepción más restringida de función *one-way*, en que se requiere entre otras cosas que la ejecución de la función inversa sea ineficiente para un alto porcentaje de las cadenas, y que los algoritmos involucrados sean probabilísticos en lugar de determinísticos.

TEMA 10.4. LA CLASE NC

Además de las clases espaciales DLOGSPACE y NLOGSPACE, existe una clase temporal que se conjetura que está incluida estrictamente en P. Se denomina NC (por *Nick's class*, o la clase de Nick, en homenaje a Nick Pippenger, quien trabajó sobre este tema), y se interpreta como la clase de los problemas de P que se pueden resolver mucho más rápido con algoritmos paralelos.

El modelo de ejecución con paralelismo más conocido en que se considera la clase NC lo constituyen las *familias polinomiales uniformes de circuitos booleanos*. Una familia de circuitos booleanos es una secuencia de circuitos booleanos c_1, c_2, …, que acepta la unión de los lenguajes reconocidos por los c_n. Todo c_n tiene n compuertas de entrada y una compuerta de salida (también se pueden contemplar varias compuertas), su tamaño (cantidad de compuertas) se denota con $H(c_n)$, y su profundidad (longitud del camino más largo desde una compuerta de entrada hasta la compuerta de salida) se denota con $T(c_n)$. Intuitivamente, el tamaño es la cantidad de procesadores, y la profundidad es el tiempo paralelo. Las familias de circuitos booleanos son polinomiales porque $H(c_n)$ es polinomial con respecto a n, restricción natural sosteniendo la analogía con la cantidad de procesadores de una computadora real. Por otro lado, que las familias sean uniformes significa que existe una MTD que computa la función $1^n \rightarrow c_n$ en espacio $O(\log H(c_n))$; la idea es que todos los circuitos booleanos representen un mismo algoritmo, para evitar inconsistencias.

Antes de caracterizar a la clase NC, cabe destacar lo siguiente en el marco del modelo de paralelismo considerado:

- Se prueba que un lenguaje pertence a P si y sólo si existe una familia polinomial uniforme de circuitos booleanos que lo reconoce, por lo que este modelo podría contribuir a probar la conjetura $P \neq NP$. De hecho, existe una conjetura que plantea que los problemas NP-completos no pueden resolverse por medio de

familias de circuitos booleanos polinomiales (uniformes o no uniformes). Incluso se demuestra que si SAT se puede resolver por circuitos booleanos polinomiales, entonces la jerarquía polinomial colapsa en el segundo nivel.

- Se define como costo o trabajo total de un circuito booleano al producto de su tamaño por su profundidad, o en otras palabras, a la cantidad de pasos ejecutados en conjunto por todos sus "procesadores". Como un algoritmo paralelo puede simularse por uno secuencial, entonces el costo total del primero no puede ser menor que el mejor tiempo del segundo. De esta manera, el paralelismo no puede resolver eficientemente ningún problema NP-completo, asumiendo P ≠ NP: costo total exponencial con tamaño polinomial implica profundidad exponencial.

La clase NC es la clase de los lenguajes que pueden reconocerse por familias polinomiales uniformes de circuitos booleanos de profundidad polilogarítmica con respecto al tamaño de las entradas. Se define como la unión infinita de subclases NC_k, cada una de las cuales se asocia a circuitos booleanos de profundidad $O(\log^k n)$. Como dijimos previamente, se asume que NC caracteriza a los problemas de decisión con solución paralela eficiente. La clase FNC se define de manera similar considerando los problemas de búsqueda. Algunos ejemplos de problemas en NC son las cuatro operaciones aritméticas elementales, el problema de la alcanzabilidad en grafos (orientados o no orientados), distintas operaciones con matrices de n^2 elementos como el producto, el determinante y la inversa, etc.

Se prueba fácilmente que $NC \subseteq P$:

- Los circuitos booleanos c_n tienen tamaño $H(c_n)$ polinomial con respecto a n, se pueden generar en espacio determinístico $O(\log H(c_n))$ por la condición de uniformidad, y así en tiempo determinístico polinomial con respecto a n.
- El problema de evaluación de circuitos booleanos (CIRCUIT VALUE) pertenece a P.

No se sabe si la inclusión es estricta. Asumiendo NC ≠ P, como se cumple que NC es cerrada con respecto a las reducciones *log-space*, entonces probar que un lenguaje L es P-completo con respecto a las reducciones *log-space* implica que $L \notin NC$. Así como asumiendo P ≠ NP se prueba que existe la clase de complejidad intermedia NPI,

asumiendo NC ≠ P pueden haber en P problemas no P-completos fuera de NC. Como ejemplo de problema P-completo ya hemos mencionado al problema CIRCUIT VALUE. Otro ejemplo clásico es el problema del flujo máximo en una red: en su versión de problema de decisión, consiste en determinar, dado un grafo con arcos con capacidades, si se puede transportar un flujo de valor al menos B desde un vértice v_1 con grado de entrada 0 hasta un vértice v_2 con grado de salida 0. Como CIRCUIT VALUE, este problema parece ser inherentemente secuencial.
También se prueba

$$NC_1 \subseteq DLOGSPACE \subseteq NLOGSPACE \subseteq NC_2$$

Por lo tanto, asumiendo $NC_1 \neq DLOGSPACE$, encontrar un lenguaje L que sea DLOGSPACE-completo con respecto a un tipo de reducción adecuada implica que L no admite una solución paralela de tiempo logarítmico. Y más en general, considerando NC ⊂ P, y por lo tanto DLOGSPACE ⊂ P, probar que un problema es P-completo con respecto a las reducciones *log-space* significa que no admite una resolución eficiente ni en términos de tiempo paralelo ni en términos de espacio determinístico logarítmico, o en otras palabras, que resolverlo requiere mucha información simultánea en memoria.
La clase NC también se puede definir en términos de las *máquinas PRAM* (por *parallel random access machines*, o máquinas de acceso aleatorio paralelo), que constituyen la versión paralela de las máquinas RAM que tratamos en el Tema 5.1. Las PRAM son máquinas sincrónicas (todos los procesadores ejecutan una instrucción al mismo tiempo y todas las instrucciones tardan lo mismo), y tienen una única memoria global compartida por todos los procesadores. Como en el caso de los circuitos booleanos, se establece una condición de uniformidad: debe existir una MTD que genere las PRAM en espacio logarítmico con respecto a la longitud de los inputs. En términos de este modelo de ejecución alternativo, más cercano a las computadoras reales, NC se define como la clase de los problemas que se resuelven en tiempo polilogarítmico mediante PRAM con una cantidad polinomial de procesadores.

Ejercicios de la Clase 10

1. Probar que las Cook-reducciones son reflexivas y transitivas.
2. Completar la prueba del Ejemplo 10.2.

3. Probar que SAT es auto-reducible.
4. Probar que los siguientes incisos son equivalentes:
   i. A es Cook-reducible a B.
   ii. A^C es Cook-reducible a B.
   iii. A es Cook-reducible a B^C.
   iv. A^C es Cook-reducible a B^C.
5. Probar que NP es cerrado con respecto a las Cook-reducciones si y sólo si NP es cerrado con respecto al complemento.
6. Probar que FSAT es FNP-completo.
7. Dada una fórmula booleana φ, sea V(φ) la función que calcula la cantidad de asignaciones de valores de verdad que satisfacen φ. Se define que A es una aproximación de una función f con respecto a una cota $C \geq 1$, si es una función que cumple que $(f(x) / C) \leq A(x) \leq (f(x) . C)$. Probar que si existe una aproximación A en P de la función V con respecto a una cota $C \geq 1$, entonces P = NP.
8. Completar la prueba del Ejemplo 10.4.
9. Probar que el problema QBF_k es efectivamente un caso particular de QBF.
10. Justificar, en el marco de la complejidad temporal, que no se pierde generalidad cuando se trabaja con MTN con grado no determinístico dos, todos los pasos son no determinísticos, y todas las computaciones ejecutan la misma cantidad de pasos.
11. El problema MAJSAT (por *majority satisfactibility*, es decir satisfactibilidad por mayoría), consiste en determinar si una fórmula booleana es satisfactible por al menos la mitad más uno de las asignaciones de valores de verdad posibles. Probar que MAJSAT ∈ PP.
12. Probar:
    i. La clase RP no cambia si se modifica la definición de las máquinas RP, reemplazando la fracción 1/2 por cualquier otra.
    ii. La clase BPP no cambia si se modifica la definición de las máquinas BPP, reemplazando la fracción 3/4 por cualquier otra que sea mayor que 1/2.
13. Probar que las clases RP y BPP son cerradas con respecto a la intersección y la unión.
14. Probar que las clases ZPP, RP, BPP y PP son cerradas con respecto a las reducciones polinomiales.
15. Probar:
    i. $P \subseteq ZPP \subseteq RP \subseteq BPP \subseteq PP$.

ii. $RP \subseteq NP \subseteq PP$.

iii. Si $NP \subseteq BPP$, entonces $NP = RP$.

16. Probar que a partir de una MTD M que trabaja en tiempo T(n), se puede construir en espacio logarítmico una familia de circuitos booleanos de tamaño $O(T^2(n))$ que reconoce L(M).
17. Considerando el ejercicio anterior, probar que CIRCUIT VALUE es P-completo con respecto a las reducciones *log-space*, como se indicó previamente. Probar además que la P-completitud de CIRCUIT VALUE sigue valiendo aún con la restricción de que las compuertas de los circuitos booleanos sean sólo de tipo $\vee$ o bien $\neg$.

Notas y bibliografía para la Parte 2

Si bien durante la década de 1950 y comienzos de la década de 1960 autores como A. Grzegorczyk y M. Rabin estudiaron clases de funciones recursivas para discutir informal o elípticamente temas de complejidad, incluso haciendo referencia a la relación entre P y NP, se considera que el estudio sistemático y formal de la complejidad temporal y espacial empezó con (Hartmanis, Lewis & Stearns, 1965) y (Hartmanis & Stearns, 1965). La idea de que el tiempo polinomial es una propiedad deseable de los problemas computables apareció por primera vez en los trabajos de varios investigadores de la teoría de la computación y la lógica, como J. von Neumann, M. Rabin, A. Cobham y J. Edmonds (particularmente, este último utilizó el término *buenos algoritmos* para caracterizar a los algoritmos polinomiales). En (Cobham, 1964), (Edmonds, 1965), (Edmonds, 1966-1967a) y (Edmonds, 1966-1967b), se manifiesta la importancia de la clase P, se estudia informalmente la clase NP, y se conjetura que el problema del viajante de comercio no tiene solución determinística polinomial.

La notación O proviene de las matemáticas, y fue propuesta en (Knuth, 1968), libro fundacional del análisis y diseño de algoritmos. En (Blum, 1967a) se establece un marco axiomático para definir medidas dinámicas de complejidad, independientes del modelo de ejecución. El mismo autor estudia en (Blum, 1967b) el tamaño de las máquinas, como representante importante de las medidas estáticas de complejidad. Distintos ejemplos de medidas estáticas de complejidad se describen en (Li & Vitanyi, 1989). Para leer sobre representaciones razonables, ver por ejemplo (Garey & Johnson, 1979). Una de las principales motivaciones para utilizar funciones tiempo-construibles es el Teorema del Hueco (*Gap Theorem*), demostrado en (Borodin, 1972). Otro de los enunciados que hemos mencionado en la Clase 6 para caracterizar a la jerarquía temporal es el Teorema de Aceleración (Speed Up Theorem), que se prueba en (Blum, 1971).

Continuamente se intentan identificar nuevos problemas dentro de las clases P y FP. En (Cormen, Leiserson & Rivest, 1990) se presenta una descripción detallada y un análisis de algoritmos significativos de estas clases, con las estructuras de datos y las técnicas de programación utilizadas.

En (Ladner, Lynch & Selman, 1975) se comparan varias formas de reducciones polinomiales, y además se considera el efecto de introducirles el no determinismo.

Los trabajos fundacionales de la NP-completitud pertenecen a S. Cook, R. Karp y L. Levin, si bien fue K. Gödel quien primero se refirió a la complejidad computacional de un problema NP-completo: en 1956, en una carta a von Neumann, le preguntaba en cuántos pasos una máquina de Turing puede decidir si existe una prueba de tamaño n para una fórmula de la lógica de primer orden (Gödel confiaba en que este problema podía ser resuelto en tiempo lineal o cuadrático). El Teorema de Cook, con el que se prueba que el problema SAT es NP-completo, apareció en (Cook, 1971); el interés de Cook se centraba en los procedimientos de prueba de los teoremas en el cálculo proposicional. (Karp, 1972) fue la publicación que más contribuyó a la difusión de la noción de NP-completitud y su importancia en la optimización combinatoria; el trabajo describe veintiún problemas NP-completos, varios de los cuales hemos considerado en este libro, probados utilizando reducciones de tiempo polinomial, sugiriendo que dichos problemas son intratables (también se describe un problema PSPACE-completo, el del reconocimiento de cadenas en los lenguajes sensibles al contexto). Independientemente, en (Levin, 1973) se mostraron varios ejemplos de problemas combinatorios identificados con la NP-completitud; Levin se proponía probar la necesidad inherente de la "fuerza bruta" para resolver determinados problemas de búsqueda. Luego de estas obras pioneras se publicó un trabajo muy completo, desde entonces referencia obligada sobre los problemas NP-completos, que fue (Garey & Johnson, 1979): se describe exhaustivamente la clase NP, sus subclases P, NPI y NPC, y distintas técnicas de reducción. El p-isomorfismo de los problemas NP-completos se estudia por ejemplo en (Berman & Hartmanis, 1977); en el mismo trabajo se conjetura además que no hay lenguajes NP-completos dispersos. En (Joseph & Young, 1985), la conjetura del p-isomorfismo se relaciona con la existencia de funciones "difíciles de invertir"; más precisamente, se plantea que si dichas funciones existen, entonces hay una clase especial de lenguajes NP-completos, conocidos como *k-creativos*, que no son p-isomorfos a SAT.

El teorema que establece la existencia de la clase NPI supeditada a la conjetura $P \neq NP$, planteado en la Clase 9, se desarrolla en (Ladner, 1975).

Para leer sobre propiedades de la jerarquía exponencial, y particularmente su relación con el no determinismo, ver por ejemplo (Hartmanis, Immerman & Sewelson, 1985).

Con respecto a la complejidad espacial, es de destacar históricamente el Teorema de Savitch (Savitch, 1970), previo al Teorema de Cook. El Teorema de Savitch, como así también el Teorema de Immerman (Immerman, 1988), son teoremas centrales que

establecen el impacto del no determinismo en la complejidad espacial; sus pruebas pueden encontrarse en la segunda parte de nuestro libro anterior (Rosenfeld & Irazábal, 2010). Muy relevante también en la evolución de la complejidad espacial fue el trabajo (Stockmeyer & Meyer, 1973), en que se describen problemas PSPACE-completos como QBF y el problema de la equivalencia de las expresiones regulares. La demostración de que el problema O-ALCANZABILIDAD es NLOGSPACE-completo con respecto a las reducciones logarítmicas figura en (Jones, 1975).

Una exhaustiva lista de problemas P-completos se puede encontrar en el trabajo (Miyano, Shiraishi & Shoudai, 1989).

Uno de los primeros trabajos sobre la aproximación de los problemas de optimización fue (Johnson, 1974).

Las relativizaciones de P vs NP se publicaron inicialmente en (Baker, Gill & Solovay, 1975). La jerarquía polinomial, también basada en las máquinas de Turing con oráculo, apareció en (Stockmeyer, 1977); se planteaba como una versión acotada de la jerarquía aritmética de funciones introducida por S. Kleene.

Los algoritmos probabilísticos más conocidos para resolver el problema de composicionalidad se publicaron en (Rabin, 1976) y (Solovay & Strassen, 1977). Los dos se basan en el concepto de testigo de composicionalidad y su abundancia. Con (Gill, 1977) se introdujeron las máquinas de Turing probabilísticas y las clases asociadas de problemas PP, BPP, RP y ZPP, con el objeto de analizar si tales máquinas pueden requerir menos tiempo o espacio que las máquinas determinísticas; se prueban numerosas propiedades de las clases probabilísticas mencionadas.

La complejidad de los circuitos booleanos se describió inicialmente en (Shannon, 1949), donde se propone como medida de complejidad el tamaño del mínimo circuito que computa una función. En (Pippenger, 1979) se introdujo la clase NC, definida como la clase de los lenguajes reconocidos por MT determinísticas de tiempo polinomial que efectúan una cantidad logarítmica de cambios de dirección del cabezal. La formulación de NC en términos de las familias uniformes de circuitos polinomiales, tal como aparece en nuestro libro, corresponde a (Cook, 1979). Para leer sobre el modelo de las máquinas PRAM, ver por ejemplo (Fortune & Wyllie, 1978).

Para profundizar en los distintos temas sobre complejidad computacional presentados en las cinco clases de la segunda parte del libro, incluyendo referencias históricas y bibliográficas, se recomienda recurrir a (Bovet & Crescenzi, 1994) y (Papadimitriou, 1994). Sugerimos también para consulta (Arora & Barak, 2009), (Balcázar, Díaz &

Gabarró, 1988), (Balcázar, Díaz & Gabarró, 1990) y (Goldreich, 2008). Otra lectura recomendada es (Gasarch, 2012). Este artículo presenta una encuesta realizada recientemente a más de ciento cincuenta referentes de la teoría de la complejidad, preguntando fundamentalmente por la relación entre las clases P y NP y con qué técnicas y en qué momento se resolverá la conjetura (en uno u otro sentido). Además se compara la encuesta con una similar realizada diez años atrás. De los resultados se destaca que el 83% opinó que $P \neq NP$ (en 2002 el porcentaje fue del 61%), y el 9% que $P = NP$ (en este caso fue el mismo guarismo que en 2002). Por otro lado, el 53% opinó que el problema se resolverá antes del año 2100 (en la encuesta anterior el porcentaje fue del 62%, lo que marca una tendencia pesimista, en contraposición a lo que se percibía en los primeros años después de los trabajos de Cook, Karp y Levin). Con respecto al modo de resolución, la mayoría consideró que se basará en técnicas hoy desconocidas, y el resto de las opiniones se repartieron esencialmente entre la lógica, la geometría computacional y la combinatoria. Todos los que consideraron que el problema se resolverá vía un algoritmo indicaron que $P = NP$, y que la prueba se encontrará más temprano que tarde. También es para remarcar que hubo más opiniones negativas que positivas en cuanto a la importancia de la computación cuántica en esta cuestión, y que la mayoría opinó que el problema del isomorfismo de grafos está en P y el de la factorización fuera de P. A propósito, recién hace unos pocos años (Agrawal, Kayal & Saxena, 2004) se probó que el problema de primalidad está en P; en verdad, en 1976 ya se había encontrado un algoritmo eficiente para resolver el problema, pero asumiendo la Hipótesis de Riemann, otro de los problemas del milenio según el Clay Mathematics Institute.

Se detallan a continuación las referencias bibliográficas mencionadas previamente:

- Agrawal, M., Kayal, N. & Saxena, N. (2004). "Primes is in P". *Annals of Mathematics,* 160(2), 781-793.
- Arora, S. & Barak, B. (2009). *Computational complexity: a modern approach*. Cambridge University Press.
- Baker, T., Gill, J. & Solovay, R. (1975). "Relativizations of the P =? NP question". *SIAM Journal on Computing*, 4, 431-442.
- Balcázar, J., Díaz, J. & Gabarró, J. (1988). *Structural Complexity 1*. Springer.
- Balcázar, J., Díaz, J. & Gabarró, J. (1990). *Structural Complexity 2*. Springer.

- Berman, L. & Hartmanis, J. (1977). "On isomorphisms and density of NP and other complete sets". *SIAM Journal on Computing*, 6, 305-322.
- Blum, M. (1967a). "A machine independent theory of the complexity of recursive functions". *Journal of ACM*, 14, 322-336.
- Blum, M. (1967b). "On the size of machines". *Information and Control*, 11, 257-265.
- Blum, M. (1971). "On effective procedures for speeding up algorithms". *Journal of ACM*, 18(2), 290-305.
- Borodin, A. (1972). "Computational complexity and the existence of complexity gaps". *Journal of ACM*, 19, 158-174.
- Bovet, D. & Crescenzi, P. (1994). *Introduction to the theory of complexity*. Prentice-Hall.
- Cobham, A. (1964). "The intrinsic computational difficulty of functions". *Proc. Congress for Logic, Mathematics, and Philosophy of Science*, 24-30.
- Cook, S. (1971). "The complexity of theorem-proving procedures". *Proceedings of the 3rd IEEE Symp. on the Foundations of Computer Science*, 151-158.
- Cook, S. (1979). "Deterministic CFL's are accepted simultaneously in polynomial time and log squared space". *Proc. ACM Symposium on Theory of Computing*, 338-345.
- Cormen, T., Leiserson, C. & Rivest, R. (1990). *Introduction to algorithms*. The MIT Press.
- Edmonds, J. (1965). "Paths, trees, and flowers". *Canad. J. Math*, 17(3), 449-467.
- Edmonds, J. (1966-1967a). "Optimum branching". *J. Res. National Bureau of Standards, Part B*, 17B(4), 233-240.
- Edmonds, J. (1966-1967b). "Systems of distinct representatives and linear algebra". *J. Res. National Bureau of Standards, Part B*, 17B(4), 241-245.
- Fortune, S. & Wyllie, J. (1978). "Parallelism in random access machines". *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, 114-118.
- Garey, M. & Johnson, D. (1979). *Computers and intractability: a guide to the theory of NP-completeness*. Freeman.

- Gasarch, W. (2012). "The second P =? NP poll". *ACM SIGACT News*, 43(2), 53-77.
- Gill, J. (1977). "Computational complexity of probabilistic Turing machines". *SIAM Journal on Computing*, 6, 675-695.
- Goldreich, O. (2008). *Computational complexity: a conceptual perspective*. Cambridge University Press.
- Hartmanis, J., Immerman, N. & Sewelson, W. (1985). "Sparse sets in NP – P: EXPTIME versus NEXPTIME". *Information and Control*, 65, 158-181.
- Hartmanis, J., Lewis, P. & Stearns, R. (1965). "Hierarchies of memory-limited computations". *Proc. 6th Annual IEEE Symp. on Switching Circuit Theory and Logic Design*, 179-190.
- Hartmanis, J. & Stearns, R. (1965). "On the computational complexity of algorithms". *Transactions of the AMS*, 117, 285-306.
- Immerman, N. (1988). "Nondeterministic space is closed under complementation". *SIAM Journal on Computing*, 17, 935-938.
- Johnson, D. (1974). "Approximation algorithms for combinatorial problems". *Journal of Computer and System Sciences*, 9, 256-278.
- Jones, N. (1975). "Space-bounded reducibility among combinatorial problems". *Journal of Computer and System Sciences*, 11, 68-85.
- Joseph, D. & Young, P. (1985). "Some remarks on witness functions for nonpolynomial and noncomplete sets in NP". *Theoretical Computer Science*, 39, 225-237.
- Karp, R. (1972). "Reducibility among combinatorial problems". *Complexity of Computer Computations, J. Thatcher & R. Miller eds., Plenum Press, New York*, 85-103.
- Knuth, D. (1968). *The art of computer programming, vol. 1: fundamental algorithms*. Addison-Wesley.
- Ladner, R. (1975). "On the structure of polynomial-time reducibility". *Journal of ACM*, 22, 155-171.
- Ladner, R., Lynch, N. & Selman, A. (1975). "A comparison of polynomial time reducibilities". *Theoretical Computer Science*, 1, 103-124.
- Levin L. (1973). "Universal sorting problems". *Problems of Information Transmission*, 9, 265-266.

- Li, M. & Vitanyi, P. (1989). "Kolmogorov complexity and its applications". *Report, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands*.
- Miyano, S., Shiraishi, S. & Shoudai, T. (1989). "A list of P-complete problems". *Technical Report RIFIS-TR-CS-17, Kyushu University*.
- Papadimitriou, C. (1994). *Computational complexity*. Addison-Wesley.
- Pippenger, N. (1979). "On simultaneous resource bounds". *Proc. IEEE Symposium on Foundations of Computer Science*, 307-311.
- Rabin, M. (1976). "Probabilistic algorithms". *Algorithms and complexity, recent results and new direction. Traub, J., ed., Academic Press*, 21-40.
- Rosenfeld, R. & Irazábal, J. (2010). *Teoría de la computación y verificación de programas*. EDULP, McGraw-Hill.
- Savitch, W. (1970). "Relationship between nondeterministic and deterministic tape complexities". *Journal of Computer and System Sciences*, 4, 177-192.
- Shannon, C. (1949). "The synthesis of two-terminal switching circuits". *Bell Systems Technical Journal*, 28, 59-98.
- Solovay, R. & Strassen,V. (1977). "A fast Monte-Carlo test for primality". *SIAM Journal on Computing*, 6, 84-85.
- Stockmeyer, L. (1977). "The polynomial-time hierarchy". *Theoretical Computer Science*, 3, 1-22.
- Stockmeyer, L. & Meyer, A. (1973). "Word problems requiring exponential time". *Proc. 5th ACM Symp. on the Theory of Computing*, 1-9.

Parte 3. Verificación de programas

Las cinco clases de la tercera y última parte del libro tratan la *verificación de programas*. Ahora los problemas se especifican formalmente mediante un lenguaje de especificación, y los algoritmos que los resuelven se escriben en un lenguaje de programación determinado. Tomando como referencia a los métodos deductivos de la lógica, el objetivo de estas clases es plantear una metodología axiomática de prueba de correctitud de un programa con respecto a una especificación (también proponer, subyacentemente, una metodología de desarrollo sistemático de programas, consistente en la construcción de un programa en simultáneo con su prueba de correctitud). Por el carácter introductorio del libro, el análisis se circunscribe a los programas secuenciales determinísticos de entrada/salida (de todos modos hacemos referencia a los programas no determinísticos y los programas concurrentes en las Clases 13 y 15, respectivamente; a los programas concurrentes les dedicamos una sección completa). La restricción también se establece en relación al dominio semántico considerado: las variables de los programas son sólo de tipo entero.

En la Clase 11 introducimos las características generales de los *métodos de verificación de programas* que consideramos hasta la Clase 15. Se describen los lenguajes que se van a utilizar: el lenguaje de programación PLW (con programas con repeticiones while), y el lenguaje de especificación Assn (con aserciones de primer orden). Se define formalmente la sintaxis y semántica de estos lenguajes. La semántica utilizada para describir PLW es la semántica operacional estructural. Se formulan las definiciones más relevantes, como estado, función semántica, aserción, precondición y postcondición, fórmula de correctitud, correctitud parcial y total de un programa, propiedades *safety* y *liveness*, sensatez y completitud de un método axiomático de verificación, etc. La especificación de programas está fuera del alcance del libro, pero de todos modos se muestran distintos ejemplos relacionados con la formulación correcta o incorrecta de especificaciones. Finalmente se introducen los métodos de verificación que se utilizarán en las clases siguientes: el método H para la prueba de correctitud parcial (dada una condición inicial o precondición, un programa si termina debe satisfacer una determinada condición final o postcondición), y el método H* para la prueba de terminación (dada una precondición, un programa debe terminar). En el marco de los programas que consideramos, la correctitud total es igual a la correctitud parcial más la

terminación, y por lo tanto con los métodos H y H* cubrimos adecuadamente lo que se necesita. La complementariedad entre los dos métodos (de naturaleza distinta, uno inductivo y el otro basado en los órdenes parciales bien fundados), se fundamenta mediante un lema conocido como el Lema de Separación.

La Clase 12 está dedicada a la *verificación de la correctitud parcial* de los programas PLW con respecto a especificaciones formuladas en el lenguaje Assn. Se describen los axiomas y reglas de inferencia que constituyen el método axiomático H, relacionados con las instrucciones atómicas y compuestas de PLW, respectivamente. Se destaca en particular la naturaleza inductiva de la regla asociada a la instrucción de repetición, centrada en el uso de una aserción invariante, y la propiedad de composicionalidad de H, según la cual la especificación de un programa depende exclusivamente de la especificación de sus subprogramas, ignorando cualquier referencia a sus estructuras internas. Se desarrollan ejemplos de pruebas utilizando H. Y se introduce una forma alternativa de documentar las pruebas, las *proof outlines*, las cuales resultan imprescindibles a la hora de verificar los programas concurrentes.

Complementando la Clase 12, la Clase 13 está dedicada a la *verificación de la terminación* de los programas PLW. Se describen los axiomas y reglas del método axiomático H*, que naturalmente, salvo la regla relacionada con la instrucción de repetición (única fuente de infinitud), coinciden con los axiomas y reglas del método H. Se muestra que en realidad H* se puede utilizar para probar directamente la correctitud total. Se formulan distintas variantes de la regla asociada a la instrucción de repetición, centradas en la noción de un variante perteneciente a un orden parcial bien fundado. Se desarrollan ejemplos de pruebas utilizando H*. Se describen las *proof outlines* de correctitud total. Y por último, se trata la verificación de la terminación de programas en un entorno de ejecución con distintas hipótesis de *fairness*, las cuales aseguran cierta equidad en lo que se refiere a la asignación de recursos computacionales (en nuestro caso, particularmente, que una instrucción habilitada infinitas o todas las veces a lo largo de una computación infinita, no puede quedar postergada perpetuamente). La noción de *fairness* no es aplicable en el paradigma de programación considerado, la tratamos por ser un concepto muy ligado a la terminación de programas (más en general, a cualquier propiedad relacionada con el progreso de las computaciones, es decir del tipo *liveness*). Por tal razón ampliamos en esta clase el espectro de los programas utilizados con los programas no determinísticos, en los que puede haber más de una computación y por lo tanto más de un estado final.

En la Clase 14 se prueba la *sensatez y completitud de los métodos de verificación* H y H*, propiedades relacionadas con la correctitud y el alcance de los métodos, respectivamente, propias de los sistemas axiomáticos de la lógica. Su cumplimiento asegura que sólo se pueden probar fórmulas de correctitud verdaderas (sensatez), y que toda fórmula verdadera se puede probar (completitud). Dado que el dominio semántico considerado es el de los números enteros, la incompletitud de nuestra metodología de verificación es insoslayable (por el Teorema de Incompletitud de Gödel), y así se introduce el concepto de completitud relativa. Se destaca el establecimiento de una condición de expresividad en el lenguaje de especificación, otra característica heredada de la lógica. Al final se analiza la sensatez y la completitud teniendo en cuenta cualquier interpretación, no sólo la interpretación (estándar) de los números enteros.

Por último, en la Clase 15 presentamos *misceláneas de verificación de programas*. En primer lugar ampliamos el lenguaje PLW con el tipo de datos arreglo, comprobamos cómo esta extensión provoca la pérdida de la sensatez de H, y planteamos una modificación para restablecer dicha propiedad. En la segunda sección se presenta un ejemplo muy simple de desarrollo sistemático de un programa PLW a partir de una especificación de Assn, guiado por los axiomas y reglas de nuestra metodología de verificación. Luego volvemos a ampliar el lenguaje PLW, esta vez con procedimientos, que incluyen recursión y parámetros. Consecuentemente, extendemos los métodos H y H* con reglas asociadas a estos mecanismos, y presentamos ejemplos. La última sección de la clase está dedicada a la verificación de los programas concurrentes, si bien este tema está fuera del alcance del libro. Se consideran sólo los programas concurrentes con memoria compartida (también conocidos como programas paralelos, para diferenciarlos de los programas distribuidos que son los que no comparten variables). Se define formalmente el lenguaje de programación a utilizar, se describe la metodología de verificación asociada, y se desarrollan ejemplos de prueba, que ahora consideran nuevas propiedades, como la ausencia de *deadlock*. Se destaca la pérdida de la composicionalidad y la incompletitud de la metodología, en este último caso por razones distintas a las esgrimidas en el marco de los programas secuenciales, y como solución se plantea, mediante *proof outlines*, un esquema de prueba en dos etapas, que incluye una regla especial para establecer la completitud. Finalmente, se introduce un segundo lenguaje de programación, con una primitiva de sincronización que permite una programación más estructurada y por lo tanto pruebas más sencillas.

Clase 11. Métodos de verificación de programas

En esta última parte del libro, nuestra mirada a los problemas decidibles se enfoca en la verificación de la correctitud de los algoritmos que los resuelven. Básicamente:

- Dejamos de utilizar el modelo de ejecución de las máquinas de Turing. Ahora recurrimos a un *lenguaje de especificación*, para describir los problemas, y a un *lenguaje de programación*, para describir los algoritmos que los resuelven, en ambos casos teniendo en cuenta una sintaxis y una semántica determinadas. De esta manera pasamos a hablar de *especificaciones* y *programas*.
- A partir de dichos componentes planteamos una *metodología de prueba* para verificar la correctitud de un programa con respecto a una especificación.

La familia de programas que consideramos es muy representativa: los programas *imperativos*. Dichos programas se caracterizan por transformar *estados* mediante las instrucciones que los componen. Por el carácter introductorio de estas clases trabajamos sólo con un paradigma de programación muy sencillo, la *programación secuencial determinística de entrada/salida* (en las Clases 13 y 15 hacemos alguna referencia a la *programación secuencial no determinística* y a la *programación concurrente*, respectivamente).

Una aproximación natural de la verificación de programas es la *operacional*, que consiste en analizar las computaciones de los programas. Esta aproximación resulta prohibitiva cuando se tratan programas concurrentes muy complejos, con numerosas computaciones y propiedades a probar.

Otra aproximación, muy difundida en la literatura y que es la que presentamos en este libro, es la *axiomática*. Tomando como marco de referencia los sistemas deductivos de la lógica, plantea un método de prueba con axiomas y reglas de inferencia asociados a las instrucciones del lenguaje de programación considerado, permitiendo probar la correctitud de un programa de la misma manera que se prueba un teorema. Las pruebas son *sintácticas*, guiadas por la estructura de los programas. Los artífices de la verificación axiomática de programas fueron fundamentalmente R. Floyd y C. Hoare, trabajando a finales de los años 1960 con diagramas de flujo y programas secuenciales determinísticos, respectivamente. J. de Bakker y P. Lauer, D. Gries y S. Owicki, y K.

Apt, W. de Roever y N. Francez, entre otros, extendieron la metodología durante los años 1970 a los programas no determinísticos, los programas concurrentes con memoria compartida y los programas distribuidos, respectivamente. Y desde comienzos de los años 1980, con pioneros como E. Clarke, E. Emerson, J. Queille y J. Sifakis, se ha venido avanzando en la verificación automática de propiedades de programas, conocida como *model checking*, en el marco de los *programas de estados finitos*, utilizando lenguajes de especificación basados en la *lógica temporal*.

En esta clase introducimos los conceptos fundamentales de la verificación axiomática de programas. Primero presentamos el lenguaje de programación y el lenguaje de especificación con los que vamos a trabajar. Luego describimos las características generales de la metodología de verificación que desarrollamos en las clases siguientes.

Definición 11.1. Lenguaje de programación PLW

El lenguaje de programación se denomina PLW (por *programming language while*, es decir, lenguaje de programación con repeticiones while). La sintaxis de PLW es la siguiente. Si S es un programa PLW, entonces tiene alguna de estas formas:

$$S :: skip \mid x := e \mid S_1 ; S_2 \mid \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ fi} \mid \text{while } B \text{ do } S_1 \text{ od}$$

Los indices en S_1 y S_2 no forman parte del lenguaje, sólo se utilizan para facilitar la definición. Las expresiones son de tipo entero (expresión e) o booleano (expresión B). Las expresiones enteras tienen la forma:

$$e :: n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 . e_2 \mid \ldots \mid \text{if } B \text{ then } e_1 \text{ else } e_2 \text{ fi}$$

donde n es una constante entera, y x es una variable entera. Las expresiones booleanas tienen la forma:

$$B :: true \mid false \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid \ldots \mid \neg B \mid B_1 \vee B_2 \mid B_1 \wedge B_2 \mid \ldots$$

Las constantes booleanas son *true* (por verdadero) y *false* (por falso). La semántica de PLW es la habitual (la definición formal la presentamos enseguida):

- La instrucción *skip* es atómica (se consume en un paso), y no tiene ningún efecto sobre las variables. Se puede usar, por ejemplo, para escribir una selección condicional sin instrucción else.
- La *asignación* x := e también es atómica, y asigna el valor de *e* a la variable x.
- La *secuencia* S_1 ; S_2, ejecuta S_1 y luego S_2.
- La *selección condicional* if B then S_1 else S_2 fi, ejecuta S_1 si B es verdadera, o S_2 si B es falsa.
- Finalmente, la *repetición* while B do S od, ejecuta S mientras B sea verdadera (toda vez, primero evalúa B y luego ejecuta eventualmente S). Cuando B es falsa, termina.

Fin de Definición

Por ejemplo, el siguiente programa PLW obtiene en una variable *y*, el factorial de una variable x > 0:

```
Sfac :: a := 1 ; y := 1 ;
        while a < x do
                a := a + 1 ; y := y . a
                        od
```

Definimos a continuación la semántica formal de PLW. Primero consideramos solamente las expresiones.

Definición 11.2. Semántica de las expresiones de PLW

La definición es inductiva, guiada por la sintaxis. Uno de los casos base corresponde a las constantes (enteras, booleanas, símbolos de función y símbolos de relación). Mediante una función I (por interpretación), a cada constante c se le asigna un valor I(c) del dominio semántico correspondiente:

- A todo numeral n, el número entero n.
- A las constantes true y false, los valores de verdad verdadero y falso, respectivamente.

- Al símbolo de suma, la función de suma habitual de dos números enteros. De manera similar se interpretan los símbolos de resta, multiplicación, etc.
- Al símbolo de negación, la función de negación de un valor de verdad. De manera similar se interpretan los símbolos de disyunción, conjunción, etc.
- Al símbolo de igualdad, la función de igualdad entre dos números enteros. De manera similar se interpretan los símbolos de menor, mayor, etc., considerando las relaciones habituales de los números enteros.

El otro caso base corresponde a las variables. A diferencia de las constantes, los valores de las variables no son fijos sino que se asignan a partir de estados. Un estado es una función σ que asigna a toda variable un valor de su tipo, en este caso un número entero (puede verse como una "instantánea" de los contenidos de las variables de un programa). La expresión σ(x) denota el contenido de la variable x según el estado σ, y el conjunto de todos los estados se denota con Σ.

La semántica del resto de las expresiones se establece a partir de la semántica de sus componentes. Formalmente, definimos la semántica de todas las expresiones de PLW mediante funciones semánticas S(e) y S(B), que a partir de un estado σ asignan valores del tipo correspondiente de la siguiente manera:

1. $S(n)(\sigma) = n$, para todo n, siendo el n de la derecha el número entero denotado por el numeral n de la izquierda.
2. S(true)(σ) = verdadero. De manera similar se define para false.
3. $S(x)(\sigma) = \sigma(x)$, para toda variable x.
4. $S(e_1 + e_2)(\sigma) = S(e_1)(\sigma) + S(e_2)(\sigma)$. De manera similar se define para la resta, la multiplicación, etc. Por su parte, S(if B then e_1 else e_2 fi)(σ) = $S(e_1)(\sigma)$ si S(B)(σ) = verdadero, o $S(e_2)(\sigma)$ si S(B)(σ) = falso.
5. $S(e_1 = e_2)(\sigma) = (S(e_1)(\sigma) = S(e_2)(\sigma))$, que es verdadero o falso. De manera similar se define para el menor, el mayor, etc.
6. $S(\neg B)(\sigma) = \neg S(B)(\sigma)$. De manera similar se define para la disyunción, la conjunción, etc.

Fin de Definición

Dado que trabajaremos con interpretaciones fijas, es decir que los dominios semánticos no van a variar (recién en la Clase 14 haremos algunas generalizaciones), podemos

abreviar $S(e)(\sigma)$ con $\sigma(e)$, y $S(B)(\sigma)$ con $\sigma(B)$, y así aplicar estados directamente sobre expresiones. Para denotar que una variable x tiene un valor particular n en un estado σ, se utiliza la expresión $\sigma[x \mid n]$, lo que se conoce como *variante de un estado*, útil para definir después la semántica de la instrucción de asignación. Formalmente, dadas dos variables x e y, se define:

- $\sigma[x \mid n](x) = n$
- $\sigma[x \mid n](y) = \sigma(y)$

A continuación completamos la definición de la semántica formal de PLW, considerando sus instrucciones. Utilizamos la *semántica operacional estructural*, desarrollada por G. Plotkin a fines de los años 1970. Las descripciones se hacen en términos de las operaciones de una máquina abstracta, estableciendo cómo un programa transforma estados a partir de un estado inicial. Más precisamente, dados un programa S y un estado inicial σ, se asocia a la *configuración inicial* $C_0 = (S, \sigma)$ una *computación* $\pi(S, \sigma)$, que es una secuencia de configuraciones $C_0 \rightarrow C_1 \rightarrow \ldots$, donde cada C_i es un par (S_i, σ_i), siendo S_i la *continuación sintáctica* (lo que le falta por consumirse al programa S), y σ_i el estado corriente. De esta manera, una computación no puede ser extendida, es *maximal*. Las computaciones $\pi(S, \sigma)$ se definen inductivamente por medio de una *relación de transición* entre configuraciones, basada en la sintaxis de S (de acá proviene la calificación de estructural de esta semántica). Las computaciones son finitas o infinitas. Una computación finita $C_0 \rightarrow \ldots \rightarrow C_k$ se abrevia con $C_0 \rightarrow^* C_k$. C_k se denomina *configuración terminal*, y es un par (E, σ_k). E (por *empty*, vacío) es la *continuación sintáctica vacía*, no forma parte del lenguaje, sólo se utiliza para indicar que el programa asociado terminó, y cumple que S ; E = E ; S = S para todo programa S. El estado final de una computación $\pi(S, \sigma)$ se denota con $val(\pi(S, \sigma))$. Si $\pi(S, \sigma)$ es infinita, se escribe $val(\pi(S, \sigma)) = \perp$, y se define que $val(\pi(S ; S', \sigma)) = \perp$ para todo programa S' (es decir que la no terminación se propaga). Al símbolo $\perp$ se lo conoce como *estado indefinido*, para diferenciarlo de los que están definidos, que se conocen como *estados propios*. Además de la expresión $val(\pi(S, \sigma))$ también se puede utilizar $M(S)(\sigma)$, siendo $M: PLW \rightarrow (\Sigma \rightarrow \Sigma)$ la función semántica asociada al lenguaje PLW. Dado un programa S, M(S) es una función total, porque cuando una computación $\pi(S, \sigma)$ no termina, se cumple $M(S)(\sigma) = \perp$.

Definición 11.3. Semántica de las instrucciones de PLW

La semántica de las instrucciones se puede establecer definiendo inductivamente una relación de transición $\rightarrow$ entre configuraciones, de la siguiente manera:

1. $(skip, \sigma) \rightarrow (E, \sigma)$
2. $(x := e, \sigma) \rightarrow (E, \sigma[x \mid \sigma(e)])$
3. Si $(S, \sigma) \rightarrow (S', \sigma')$, entonces para todo T de PLW: $(S ; T, \sigma) \rightarrow (S' ; T, \sigma')$
4. Si $\sigma(B)$ = verdadero, entonces (if B then S_1 else S_2 fi, σ) $\rightarrow (S_1, \sigma)$
   Si $\sigma(B)$ = falso, entonces (if B then S_1 else S_2 fi, σ) $\rightarrow (S_2, \sigma)$
5. Si $\sigma(B)$ = verdadero, entonces (while B do S od, σ) $\rightarrow$ (S ; while B do S od, σ)
   Si $\sigma(B)$ = falso, entonces (while B do S od, σ) $\rightarrow (E, \sigma)$

Fin de Definición

De ahora en más abreviamos $\sigma[x \mid \sigma(e)]$ con $\sigma[x \mid e]$. Como se aprecia en la definición:

- El skip no modifica el estado inicial y se consume en un paso.
- La asignación modifica eventualmente el estado inicial y también se consume en un paso.
- Como la continuación sintáctica E cumple que S ; E = E ; S = S, si $val(\pi(S, \sigma)) = \sigma' \neq \perp$, entonces $(S ; T, \sigma) \rightarrow^* (T, \sigma')$.
- La evaluación de una expresión booleana, tanto en la selección condicional como en la repetición, consume un solo paso y no modifica el estado corriente.

De la definición se infiere el determinismo de PLW: un programa tiene una sola computación, sólo una configuración sucede a otra. Además, se pueden desarrollar fácilmente las distintas formas que pueden tener las computaciones de los programas. Por ejemplo, en el caso de la secuencia, una computación $\pi(S_1 ; S_2, \sigma)$ tiene tres formas posibles:

- Una computación infinita $(S_1 ; S_2, \sigma_0) \rightarrow (T_1 ; S_2, \sigma_1) \rightarrow (T_2 ; S_2, \sigma_2) \rightarrow \ldots$, cuando S_1 no termina a partir de σ_0.

- Otra computación infinita $(S_1 ; S_2, \sigma_0) \rightarrow \ldots \rightarrow (S_2, \sigma_1) \rightarrow (T_1, \sigma_2) \rightarrow (T_2, \sigma_3) \rightarrow \ldots$, cuando S_1 termina a partir de σ_0 y S_2 no termina a partir de $val(\pi(S_1, \sigma_0)) = \sigma_1$.
- Una computación finita $(S_1 ; S_2, \sigma_0) \rightarrow \ldots \rightarrow (S_2, \sigma_1) \rightarrow \ldots \rightarrow (E, \sigma_2)$, cuando S_1 termina a partir de σ_0 y S_2 termina a partir de $val(\pi(S_1, \sigma_0)) = \sigma_1$.

De modo similar se pueden desarrollar las formas de las computaciones del resto de las instrucciones (queda como ejercicio). En el ejemplo siguiente mostramos la forma de la computación de un programa PLW muy simple.

Ejemplo 11.1. Forma de la computación del programa S_{swap}

Sea el programa S_{swap} :: z := x ; x := y ; y := z, y el estado inicial σ_0, con $\sigma_0(x) = 1$ y $\sigma_0(y) = 2$. Vamos a probar que S_{swap} intercambia los contenidos de las variables x e y, desarrollando la computación asociada en base a la definición de la relación $\rightarrow$:

$\pi(S_{swap}, \sigma_0) = (z := x ; x := y ; y := z, \sigma_0[x \mid 1][y \mid 2]) \rightarrow$
$(x := y ; y := z, \sigma_0[x \mid 1][y \mid 2][z \mid x]) = (x := y ; y := z, \sigma_0[x \mid 1][y \mid 2][z \mid 1]) \rightarrow$
$(y := z, \sigma_0[x \mid 1][y \mid 2][z \mid 1][x \mid y]) = (y := z, \sigma_0[y \mid 2][z \mid 1][x \mid 2]) \rightarrow$
$(E, \sigma_0[y \mid 2][z \mid 1][x \mid 2][y \mid z]) = (E, \sigma_0[z \mid 1][x \mid 2][y \mid 1])$

Por lo tanto, $val(\pi(S_{swap}, \sigma_0)) = \sigma_1$, con $\sigma_1(x) = 2$ y $\sigma_1(y) = 1$. Generalizando, se cumple que si $\sigma_0(x) = X$ y $\sigma_0(y) = Y$, entonces $val(\pi(S_{swap}, \sigma_0)) = \sigma_1$, con $\sigma_1(x) = Y$ y $\sigma_1(y) = X$.

Fin de Ejemplo

Notar en el ejemplo que las expresiones de la forma $\sigma[x \mid a][x \mid b]$ se reemplazaron por $\sigma[x \mid b]$. Queda como ejercicio demostrar la igualdad de ambas expresiones.

Para simplificar la presentación no contemplamos la posibilidad de *falla* de un programa PLW, es decir que termine incorrectamente, por ejemplo por la ejecución de una división por cero (asumimos convenciones del tipo $\sigma(x / 0) = 0$ y $\sigma(z \bmod 0) = \sigma(z)$). Otro ejemplo de falla es una violación de rango si se utilizan arreglos. Si se incluyera la posibilidad de falla se debería considerar un estado de falla f, y definir que si $val(\pi(S, \sigma)) = f$, entonces $val(\pi(S ; S', \sigma)) = f$ para todo programa S' (es decir que la falla se propaga).

Como alternativa a la semántica operacional se puede utilizar la *semántica denotacional*, desarrollada a lo largo de los años 1970 por M. Gordon, D. Scott, J. Stoy y C. Strachey, entre otros. La idea básica de esta aproximación es definir un apropiado dominio semántico para calcular la función semántica M(S) en base a la estructura de S, utilizando una técnica conocida como *punto fijo* para tratar con las repeticiones y la recursión. Se utilizan distintas funciones para describir el significado o valor de las construcciones del lenguaje, de manera tal que el 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). Notar que así especificamos la semántica de las expresiones de PLW. Por lo tanto, la semántica denotacional tiene un enfoque más matemático que la semántica operacional. El problema de esta aproximación es que se complica sobremanera cuando se consideran programas concurrentes muy complejos.

Existen además las alternativas *axiomática* y *algebraica* para definir la semántica formal de un lenguaje de programación. En el primer caso, se recurre a la propia aproximación axiomática de Hoare para la verificación de programas. Con respecto a la semántica algebraica, la misma fue desarrollada en los años 1980 por J. Goguen entre otros, y es similar a la semántica denotacional, también se basa en funciones que asignan significado a cada construcción del lenguaje considerado, pero ahora tomando como referencia los conceptos del álgebra universal.

Habiendo completado la presentación del lenguaje de programación PLW, ahora pasamos a describir el lenguaje de especificación con el que vamos a trabajar.

Definición 11.4. Lenguaje de especificación Assn

El lenguaje de especificación se denomina Assn (por *assertions*, es decir *aserciones*, dado que con este lenguaje formulamos aserciones sobre estados). Assn es un lenguaje de primer orden, interpretado sobre el dominio de los números enteros. La interpretación es la estándar, es decir la de los números enteros como los conocemos, considerando sus operaciones y relaciones habituales. Las aserciones tienen la forma:

$$p :: true \mid false \mid e_1 = e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid \ldots \mid \neg p \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \mid \ldots \mid \exists x\colon p \mid \forall x\colon p$$

Como se oberva, toda expresión booleana es una aserción. Presentamos a continuación la semántica de las aserciones. Se utilizan funciones semánticas S(p), que se definen

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

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

Fin de Definición

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

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

También se define, naturalmente, que ninguna aserción p es verdadera cuando no hay terminación, es decir:

- ⊥ |≠ p, para toda aserción p.

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

- var(S): el conjunto de las variables de un programa S.
- change(S): el conjunto de las variables modificables de un programa S, es decir, las variables que aparecen en la parte izquierda de las asignaciones de S.
- var(e): el conjunto de las variables de una expresión e.

- free(p): el conjunto de las variables libres de una aserción p, es decir, las variables de p no alcanzadas por ningún cuantificador.

Cuando tratemos la verificación de la instrucción de asignación de PLW, nos valdremos del mecanismo de *sustitución* de la lógica. 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. Por ejemplo, si p es la aserción

$$\exists z: x \neq z \wedge \forall x: \neg(x > u)$$

entonces p[x | v] resulta

$$\exists z: v \neq z \wedge \forall x: \neg(x > u)$$

Hay que tener cuidado cuando las variables que sustituyen a otras no están libres en las aserciones. Por ejemplo, la sustitución p[x | z] no puede aplicarse de la siguiente manera:

$$\exists z: z \neq z \wedge \forall x: \neg(x > u)$$

Esta aserción es falsa. El mecanismo de sustitución establece en este caso que antes debe renombrarse la variable cuantificada (en el ejemplo, la variable z) con una nueva variable. De esta manera, utilizando una variable w, la aserción p[x | z] resulta

$$\exists w: z \neq w \wedge \forall x: \neg (x > u)$$

Por su parte, e[x | e'] denota la sustitución en la expresión e de todas las ocurrencias de la variable x por la expresión e'.
Una importante propiedad de las sustituciones se formula en el Lema de Sustitución, al cual recurriremos más adelante. Establece que

$$\sigma[x \mid e] \models p \leftrightarrow \sigma \models p[x \mid e]$$

La prueba del lema se basa directamente en la definición de sustitución, y queda como ejercicio.

Una especificación Φ de un programa S de PLW se va a expresar como un par de aserciones (p, q), correspondientes a la entrada y la salida de S, respectivamente. Dada Φ = (p, q), la aserción p se denomina *precondición* y denota el conjunto de estados iniciales de S, y la aserción q se denomina *postcondición* y denota el conjunto de estados finales de S. De esta manera, una especificación establece la relación que debe existir entre los estados iniciales y los estados finales de un programa. Por ejemplo, la especificación

$$\Phi = (x = X, x = 2X)$$

es satisfecha por cualquier programa que duplica su entrada x. En este caso x es una *variable de programa*, mientras que X se conoce como *variable de especificación* (también se la llama *variable lógica*). Las variables de especificación sirven para "congelar" contenidos de variables de programa, las cuales pueden tener distintos contenidos al comienzo y al final. Por no ser variables de programa, las variables de especificación no pueden ser accedidas ni modificadas. Notar en el ejemplo que el conjunto de estados iniciales denotado por la precondición p = (x = X) es todo Σ. Si por ejemplo z es una variable de un programa que satisface Φ, el hecho de no mencionarla en la precondición significa que inicialmente puede tener cualquier valor. Supongamos ahora que se pretende especificar un programa que termine con la condición x > y. Una posible especificación es

$$\Phi_1 = (x = X \wedge y = Y, x > y)$$

y otra más simple, que muestra la utilidad de la aserción true, es

$$\Phi_2 = (\text{true}, x > y)$$

Las variables de especificación están implícitamente cuantificadas universalmente. Por ejemplo, en la especificación

$$\Phi = (x = X, x = X + 1)$$

satisfecha por todo programa que incrementa en uno su entrada x, la variable X puede tener cualquier valor. Si una variable de especificación X aparece libre en la precondición y la postcondición y se instancia con una expresión, ésta no debe incluir variables de programa, para asegurar que se instancie siempre el mismo valor. Por ejemplo, es erróneo instanciar la especificación anterior Φ de la siguiente manera:

$$\Phi' = (x = x, x = x + 1)$$

La especificación Φ' no es satisfecha por ningún programa. Sucede que los valores de x en Φ a la izquierda y a la derecha son distintos. El último ejemplo que presentamos por ahora muestra cómo aún en casos muy sencillos, se pueden plantear especificaciones erróneas (por el empleo de aserciones demasiado débiles, por ignorar que las variables de entrada se pueden modificar a lo largo de la ejecución de un programa, etc).

Ejemplo 11.2. Especificaciones erróneas

Se pretende especificar un programa que termine con y = 1 o y = 0, según al comienzo valga o no, respectivamente, una condición p(x). Una primera versión podría ser

$$\Phi_1 = (\text{true}, (y = 1 \rightarrow p(x)) \wedge (y = 0 \rightarrow \neg p(x)))$$

Pero S_1 :: y := 2 satisface Φ_1 y no es el programa pretendido. Una segunda versión de la especificación, reforzando la postcondición, podría ser

$$\Phi_2 = (\text{true}, (0 \leq y \leq 1) \wedge (y = 1 \rightarrow p(x)) \wedge (y = 0 \rightarrow \neg p(x)))$$

Pero también es errónea: dado S_2 :: x := 5 ; y := 1, si no vale p(7), vale p(5), y al comienzo x = 7, entonces S_2 satisface Φ_2 y tampoco es el programa buscado. Finalmente, utilizando una variable de especificación para conservar el valor inicial de x, llegamos a una especificación correcta:

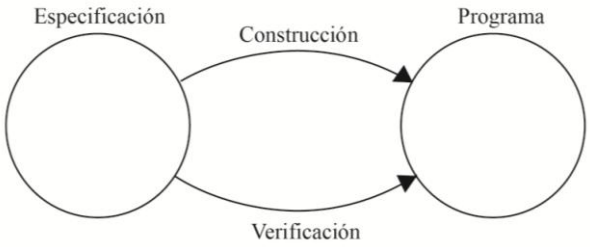
$$\Phi_3 = (x = X, (0 \leq y \leq 1) \wedge (y = 1 \rightarrow p(X)) \wedge (y = 0 \rightarrow \neg p(X)))$$

Fin de Ejemplo

Especificar es una tarea fundamental en el desarrollo de programas (su estudio está fuera del alcance de este libro). Sólo con una especificación formal se puede verificar la correctitud de un programa, comparando dos objetos provenientes de dos dominios formales, el lenguaje de especificación y el lenguaje de programación. En caso contrario, la tarea de contrastación se conoce como *validación*, en que el *testing* es la práctica más representativa (y como formuló E. Dijkstra, pionero del desarrollo sistemático de programas, el testing asegura la presencia de errores pero no su ausencia).

Hemos elegido un lenguaje de especificación muy sencillo, para focalizarnos en la problemática de la verificación de programas. En las pruebas de las próximas clases las especificaciones ya estarán elaboradas, y sólo se referirán al comportamiento funcional de entrada/salida de los programas. A propósito, el lenguaje de especificación Assn no es adecuado para expresar propiedades de programas *reactivos*. En dichos programas no hay noción de terminación, porque interactúan perpetuamente con el entorno de ejecución. Un caso típico lo constituyen los sistemas operativos. Para especificar esta familia de programas es común recurrir a lenguajes de la lógica temporal, con fórmulas del tipo "siempre se cumple la condición p", "alguna vez en el futuro se cumple la condición q", etc. La lógica temporal, además, permite expresar condiciones de *fairness* (equidad), como por ejemplo la de que en una computación infinita, una instrucción siempre habilitada no puede postergarse indefinidamente (hacemos alguna mención al *fairness* en la Clase 13).

Presentados los lenguajes PLW y Assn, estamos en condiciones de plantear en términos de ambos componentes, las características generales de la metodología de verificación de programas que desarrollamos en las próximas clases. La exposición se basa en pruebas *a posteriori*, es decir en pruebas de programas ya construidos, pero nuestro objetivo es que se perciba la metodología como un conjunto de principios generales de construcción de programas correctos, con axiomas y reglas que guían su desarrollo sistemático (en la Clase 15 ejemplificamos esta idea). Dicho de otro modo, la idea es que la metodología presentada sea vista como soporte para la construcción de un programa en simultáneo con su prueba de correctitud, como lo ilustra la figura siguiente:



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

Definición 11.5. Correctitud parcial y total de un programa

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

$$(\sigma \models p \ \wedge \ val(\pi(S, \sigma)) \neq \perp) \rightarrow val(\pi(S, \sigma)) \models q$$

En palabras, S es parcialmente correcto con respecto a $\Phi = (p, q)$, o directamente (p, q), si y sólo si a partir de cualquier estado σ que satisface la precondición p, si S termina lo hace en un estado σ' que satisface la postcondición q.

De esta manera, la correctitud parcial de S con respecto a (p, q) no se viola si existe un estado σ que satisface p a partir del cual S no termina.

Justamente, el término *parcialmente correcto* se debe a que la función semántica M(S) puede asignar a determinados estados iniciales propios el estado indefinido $\perp$.

La correctitud parcial tampoco se viola cuando se consideran estados que no satisfacen p, independientemente de lo que suceda al final de las computaciones correspondientes.

La expresión $\models \{p\}\ S\ \{q\}$ denota que S es parcialmente correcto con respecto a (p, q).

La fórmula de correctitud $\{p\}\ S\ \{q\}$ se conoce también como *terna de Hoare de correctitud parcial*.

Ya dijimos, por otro lado, que la correctitud total reúne a la correctitud parcial y la terminación. Formalmente, se define que un programa S es *totalmente correcto* con respecto a una especificación (p, q), si y sólo si para todo estado σ se cumple

$$\sigma \models p \rightarrow (val(\pi(S, \sigma)) \neq \perp \ \wedge \ val(\pi(S, \sigma)) \models q)$$

Es decir, S es totalmente correcto con respecto a (p, q), si y sólo si a partir de cualquier estado σ que satisface la precondición p, S termina en un estado σ' que satisface la postcondición q (ahora S es *totalmente correcto* con respecto a (p, q) porque M(S) asigna a todo estado inicial propio un estado propio).

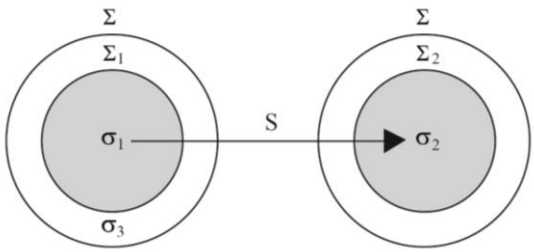
La correctitud total no se viola cuando se consideran estados que no satisfacen p, independientemente de lo que suceda al final de las computaciones correspondientes.

La expresión $\models \langle p \rangle\ S\ \langle q \rangle$ denota que S es totalmente correcto con respecto a (p, q).

La fórmula de correctitud $\langle p \rangle\ S\ \langle q \rangle$ se conoce también como *terna de Hoare de correctitud total*.

Fin de Definición

La figura siguiente ilustra qué significa que un programa S sea totalmente correcto con respecto a una especificación (p, q):



Si Σ_1 es el conjunto de estados iniciales denotado por la precondición p, y Σ_2 es el conjunto de estados finales denotado por la postcondición q, entonces, a partir de cualquier estado σ_1 de Σ_1, S debe terminar en un estado σ_2 de Σ_2. No se establece ninguna condición para S a partir de un estado σ_3 fuera de Σ_1.

El siguiente ejemplo sirve como ejercitación en el empleo de las definiciones de correctitud parcial y total.

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

La fórmula de correctitud parcial $\models \{true\}\ S\ \{true\}$ es verdadera cualquiera sea el programa S. En palabras, a partir de un estado σ, todo programa S, si termina, lo hace naturalmente en algún estado σ'. Dados S y σ, debe cumplirse entonces que

$(\sigma \models \text{true} \wedge \text{val}(\pi(S, \sigma)) \neq \perp) \rightarrow \text{val}(\pi(S, \sigma)) \models \text{true}$

La prueba es la siguiente:

- Se cumple $\sigma \models \text{true}$.
- Si $\text{val}(\pi(S, \sigma)) = \perp$, entonces la implicación vale trivialmente.
- Si $\text{val}(\pi(S, \sigma)) \neq \perp$, entonces $\text{val}(\pi(S, \sigma)) = \sigma' \models \text{true}$.

En cambio, la fórmula de correctitud total $\models \langle \text{true} \rangle\ S\ \langle \text{true} \rangle$ es verdadera sólo en los casos en que S termina cualquiera sea el estado inicial. Y por el contrario, la fórmula de correctitud parcial $\models \{\text{true}\}\ S\ \{\text{false}\}$ es verdadera sólo en los casos en que S no termina a partir de ningún estado. Queda como ejercicio probar estos dos últimos enunciados.

Fin de Ejemplo

Correspondientemente con las dos propiedades básicas mencionadas, vamos a estudiar dos métodos de verificación de programas, uno para probar la correctitud parcial y otro para probar la terminación. Esta división no es caprichosa, se debe a que las dos pruebas se basan en técnicas distintas. La prueba de correctitud parcial es inductiva, mientras que la terminación se demuestra a partir de una *relación de orden parcial bien fundada*, es decir, a partir de un orden parcial estricto sin cadenas descendentes infinitas. En este último caso utilizaremos directamente la relación menor en el conjunto de los números naturales, es decir (N, <), si bien podríamos recurrir a cualquier orden parcial bien fundado (por ejemplo (C, <), siendo C un subconjunto de N). Más en general, propiedades como la correctitud parcial, la ausencia de *deadlock* y la exclusión mutua, pertenecen a la familia de propiedades conocidas como de tipo *safety* (seguridad), que se prueban inductivamente en base a *aserciones invariantes*, y propiedades como la terminación y la ausencia de inanición, pertenecen a la familia de propiedades conocidas como de tipo *liveness* (progreso o vivacidad), que se prueban en base a *variantes* decrecientes de órdenes parciales bien fundados.

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

Para todo S, p, q: $\models \langle p \rangle\ S\ \langle q \rangle \leftrightarrow (\models \{p\}\ S\ \{q\} \wedge \models \langle p \rangle\ S\ \langle true \rangle)$

Es decir, para probar la fórmula $\langle p \rangle\ S\ \langle q \rangle$, que denota la correctitud total de S con respecto a (p, q), probaremos $\{p\}\ S\ \{q\}$, que denota la correctitud parcial de S con respecto a (p, q), y $\langle p \rangle\ S\ \langle true \rangle$, que denota la terminación de S con respecto a la precondición p. La demostración del lema queda como ejercicio.

En la Clase 12 describimos el método, denominado H, para probar la correctitud parcial de los programas PLW. Vamos a denotar con la expresión

$$\vdash_H \{p\}\ S\ \{q\}$$

que $\{p\}\ S\ \{q\}$ se prueba en H. H está compuesto por axiomas y reglas correspondientes a las instrucciones de PLW. Como en el cálculo de predicados, una prueba en H tiene la forma

1. $\vdash_H f_1$
2. $\vdash_H f_2$

……

k. $\vdash_H f_k$

tal que f_i es un axioma o es una fórmula que se obtiene de $f_1, \ldots, f_{i-1}$ por empleo de una o más reglas. En particular, f_k es la fórmula $\{p\}\ S\ \{q\}$, y así, manteniendo la analogía con la lógica, es un teorema de H.

En la Clase 13 describimos el método, denominado H*, para probar la terminación de los programas PLW. Vamos a denotar con la expresión

$$\vdash_{H^*} \langle p \rangle\ S\ \langle true \rangle$$

que $\langle p \rangle\ S\ \langle true \rangle$ se prueba en H*. En realidad, veremos que H* permite probar directamente fórmulas $\langle p \rangle\ S\ \langle q \rangle$. Naturalmente, las características de las pruebas en H* son las mismas que en H. Los axiomas y reglas también son los mismos, salvo la regla asociada a la instrucción de repetición, que es la única que puede provocar no terminación.

En la Clase 14 probamos que H y H* son *sensatos*, propiedad indispensable relacionada con la correctitud de los métodos, que asegura que sólo se prueban fórmulas de correctitud verdaderas. En la misma clase probamos también la propiedad recíproca, que H y H* son *completos* (de una manera relativa, como veremos), propiedad deseable relacionada con el alcance de los métodos, que asegura que se prueban todas las fórmulas de correctitud verdaderas.

Definición 11.6. Sensatez y completitud de un método de verificación

Considerando genéricamente un método D de verificación de correctitud parcial de programas, se define:

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

$$\vdash_D \{p\}\ S\ \{q\} \rightarrow \models \{p\}\ S\ \{q\}$$

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

$$\models \{p\}\ S\ \{q\} \rightarrow \vdash_D \{p\}\ S\ \{q\}$$

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

Fin de Definición

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

Ejercicios de la Clase 11

1. Completar la definición semántica de las expresiones de PLW.
2. Probar que una computación finita de un programa PLW termina en una configuración terminal.
3. Desarrollar las distintas formas de las computaciones de las instrucciones de PLW (en la Clase 11 sólo se mostraron las de la instrucción de secuencia).

4. Supóngase que se agregan al lenguaje PLW las instrucciones if B then S fi, y repeat S until B, con la semántica habitual. Definir formalmente la semántica de ambas instrucciones.
5. Supóngase ahora que se agregan otras dos nuevas instrucciones a PLW. La primera es S_1 or S_2, que consiste en la ejecución de S_1 o bien S_2. La segunda es S_1 and S_2, que consiste en la ejecución de S_1 seguida de la ejecución de S_2, o bien la ejecución de S_2 seguida de la ejecución de S_1. Definir formalmente la semántica de ambas instrucciones.
6. Definir la función semántica M para todos los casos de S ∈ PLW, y luego probar:
   i. $M(S_1 ; (S_2 ; S_3)) = M((S_1 ; S_2) ; S_3)$.
   ii. M(if B then S_1 else S_2 fi) = M(if not B then S_2 else S_1 fi).
   iii. M(while B do S od) = M(if B then S ; while B do S od else skip fi).
7. Se define que para todo par de programas S_1 y S_2, $S_1 = S_2$ si y sólo si $M(S_1)(\sigma) = M(S_2)(\sigma)$ cualquiera sea el estado σ. Probar que $S_1 = S_2$ si y sólo si para todo par de aserciones p y q: $\models \{p\}\ S_1\ \{q\} \leftrightarrow \models \{p\}\ S_2\ \{q\}$.
8. Sea la siguiente definición alternativa de la semántica de PLW. A cada par (S, σ) se le asocia una secuencia de estados denotada con la expresión com(S, σ). La operación de concatenación de secuencias de estados se denota con el símbolo **.**. La secuencia de estados $\sigma_1 \ldots \sigma_n$ satisface p (es decir, $\sigma_1 \ldots \sigma_n \models p$), si y sólo si $\sigma_n \models p$. Además:
   1. com(skip)(σ) = σ.
   2. com(x := e)(σ) = σ[x | e].
   3. $com(S_1 ; S_2)(\sigma) = com(S_1)(\sigma)$ **.** $com(S_2)(\sigma')$, siendo σ' el último estado de $com(S_1)(\sigma)$.

   Se pide completar la definición de com(S, σ), para los casos if B then S_1 else S_2 fi, y while B do S od.
9. Probar que para todo estado σ se cumple σ[x | a][x | b] = σ[x | b].
10. Completar la definición semántica de las aserciones de Assn.
11. Dar un ejemplo en que $p[x \mid e_1][y \mid e_2]$ no es equivalente a $p[y \mid e_2][x \mid e_1]$.
12. Probar el Lema de Sustitución, es decir, $\sigma[x \mid e] \models p \leftrightarrow \sigma \models p[x \mid e]$.
13. Definir inductivamente los conjuntos de variables var(S), change(S), var(e) y free(p).

14. Especificar un programa que calcule la raíz cuadrada entera de un número natural x. Se debe establecer que al final x tenga el mismo valor que al principio. ¿Se puede expresar también que x no se modifique a lo largo de todo el programa?
15. Determinar si las especificaciones (x = X, x = 2X) y (true, $\exists$X: x = 2X) son equivalentes.
16. Asumiendo $\models \{p\}\ S\ \{q\}$, determinar si son correctos los incisos siguientes:
    i. Si S termina en un estado final que satisface q, entonces el estado inicial satisface p.
    ii. Si S termina en un estado final que no satisface q, entonces el estado inicial no satisface p.
    iii. Si S no termina, entonces el estado inicial no satisface p.
17. Determinar lo mismo que en el ejercicio anterior, pero ahora asumiendo $\models \langle p\rangle\ S\ \langle q\rangle$.
18. Si se cumple $\models \{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}$, ¿se cumple $\models \{p_1\}\ S\ \{q_1\}$ o $\models \{p_2\}\ S\ \{q_2\}$?
19. Probar:
    i. Si free(p) $\cap$ change(S) = $\varnothing$, entonces $\models \{p\}\ S\ \{p\}$.
    ii. $\models \langle true\rangle\ S\ \langle true\rangle$ se cumple sólo cuando S termina a partir de todo estado.
    iii. $\models \{true\}\ S\ \{false\}$ se cumple sólo cuando S no termina a partir de ningún estado.
    iv. Lema de Separación: para todo S, p, q, $\models \langle p\rangle\ S\ \langle q\rangle \leftrightarrow (\models \{p\}\ S\ \{q\} \wedge \models \langle p\rangle\ S\ \langle true\rangle)$.
20. Sea D un método de verificación sensato, y supóngase que se cumple tanto $\vdash_D \{p\}\ S\ \{q\}$ como $\vdash_D \{p\}\ S\ \{\neg q\}$. ¿Qué se puede afirmar sobre S con respecto a p?

Clase 12. Verificación de la correctitud parcial

En esta clase describimos el método de verificación H, con el que se puede probar la correctitud parcial de los programas PLW.

Dada una fórmula {p} S {q}, siendo S un programa de PLW y (p, q) una especificación de S constituida por un par de aserciones de Assn, con H se puede encarar el desarrollo de una prueba de {p} S {q}, lo que se denota con la expresión $\vdash_H \{p\}\ S\ \{q\}$. Como H es sensato, si se cumple $\vdash_H \{p\}\ S\ \{q\}$ entonces también se cumple $\models \{p\}\ S\ \{q\}$, es decir que con una prueba sintáctica se prueba en realidad que S es parcialmente correcto con respecto a (p, q): dado cualquier estado σ que satisface la precondición p, si S termina a partir de σ entonces lo hace en un estado σ' que satisface la postcondición q. En cambio, nada se puede decir acerca de la terminación de S.

El método H es *composicional*, adhiere al siguiente principio: si S está compuesto por subprogramas $S_1, \ldots, S_n$, que se cumpla {p} S {q} depende solamente de que se cumplan determinadas fórmulas $\{p_1\}\ S_1\ \{q_1\}, \ldots, \{p_n\}\ S_n\ \{q_n\}$, sin ninguna referencia a la estructura interna de los S_i. Veremos después que cuando se trata con programas concurrentes, la propiedad de composicionalidad se pierde.

A continuación presentamos los componentes de H.

Definición 12.1. Axiomas y reglas del método H

1. Axioma del skip (SKIP) $\quad \{p\}\ skip\ \{p\}$
2. Axioma de la asignación (ASI) $\quad \{p[x \mid e]\}\ x := e\ \{p\}$
3. Regla de la secuencia (SEC)
$$\frac{\{p\}\ S_1\ \{r\}\ ,\ \{r\}\ S_2\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ \{q\}}$$
4. Regla del condicional (COND)
$$\frac{\{\ p \wedge B\}\ S_1\ \{q\}\ ,\ \{p \wedge \neg B\}\ S_2\ \{q\}}{\{p\}\ \text{if B then } S_1 \text{ else } S_2 \text{ fi}\ \{q\}}$$
5. Regla de la repetición (REP)
$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\ \text{while B do S od}\ \{p \wedge \neg B\}}$$
6. Regla de consecuencia (CONS)
$$\frac{p \rightarrow p_1\ ,\ \{p_1\}\ S\ \{q_1\}\ ,\ q_1 \rightarrow q}{\{p\}\ S\ \{q\}}$$

Fin de Definición

El axioma SKIP establece que si una aserción se cumple antes de la ejecución de un skip, sigue valiendo después.
El axioma ASI establece que si se cumple p en términos de x después de la ejecución de una asignación x := e, significa que antes de la ejecución se cumple p en términos de e. Por ejemplo, con ASI se puede probar

$$\{x + 1 \geq 0\}\ x := x + 1\ \{x \geq 0\}$$

Notar que ASI se lee "hacia atrás", de derecha a izquierda, lo que impone una forma de desarrollar las pruebas de H en el mismo sentido, de la postcondición a la precondición. Si bien resulta más natural plantear un axioma que se lea "hacia adelante", como

$$\{true\}\ x := e\ \{x = e\}$$

el axioma no sirve, es falso cuando la expresión *e* incluye a la variable x. Por ejemplo, si e = x + 1, se obtendría la fórmula falsa

$$\{true\}\ x := x + 1\ \{x = x + 1\}$$

El problema radica en 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 se refiere a la variable luego de la asignación. Existe de todos modos una forma correcta de ASI que se lee de izquierda a derecha, justamente diferenciando las partes derecha e izquierda de la postcondición:

$$\{p\}\ x := e\ \{\exists z: p[x \mid z] \wedge x = e[x \mid z]\}$$

Utilizaremos el axioma ASI original porque es más simple (además es la forma más difundida en la literatura).
La regla SEC establece que del cumplimiento de $\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$ se deriva el cumplimiento de $\{p\}\ S_1\ ;\ S_2\ \{q\}$. La aserción r actúa como nexo para probar la secuencia de S_1 con S_2 y luego se descarta, es decir que no se propaga a lo largo de la prueba. Podemos utilizar directamente la siguiente generalización:

$$\frac{\{p\}\ S_1\ \{r_1\}\ ,\ \{r_1\}\ S_2\ \{r_2\}\ ,\ \ldots\ ,\ \{r_{n-1}\}\ S_n\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ ;\ \ldots\ ;\ S_n\ \{q\}}$$

Queda como ejercicio demostrar que esta forma se obtiene de la regla original.

La regla COND impone una manera de verificar una selección condicional estableciendo un único punto de entrada y un único punto de salida, correspondientes a la precondición p y la postcondición q, respectivamente. A partir de p, se cumpla o no la condición B de la instrucción, debe darse que luego de las ejecuciones respectivas de S_1 y S_2 se cumple q.

La regla REP se centra en una aserción invariante p, que debe cumplirse al comienzo de un while y luego de toda iteración del mismo. Mientras valga la condición B del while, la ejecución del cuerpo S debe preservar p, y por eso al terminar la instrucción (si termina) se cumple $p \wedge \neg B$. Claramente REP no asegura la terminación, y su forma muestra que la correctitud parcial es una propiedad que se prueba inductivamente (la inducción involucrada se conoce como *computacional*).

Finalmente, la regla CONS facilita las pruebas permitiendo reforzar las precondiciones y debilitar las postcondiciones de las fórmulas de correctitud. Por ejemplo, si $r \rightarrow p$ y $\{p\}\ S\ \{q\}$, entonces por CONS se prueba $\{r\}\ S\ \{q\}$. En particular, se pueden modificar fórmulas con aserciones equivalentes. CONS es una regla especial, no depende de PLW sino del dominio semántico, en este caso los números enteros. Es una regla semántica más que sintáctica. Actúa como interface entre las fórmulas de correctitud y las aserciones verdaderas sobre los números enteros. Por el uso de CONS, algunos pasos de una prueba en H pueden ser directamente aserciones. Llamando Tr (por *true*, verdadero) al conjunto de todas las aserciones verdaderas sobre los enteros, considerando la interpretación estándar como establecimos en la clase anterior, una prueba en H se debe entender entonces como

$$Tr \vdash_H \{p\}\ S\ \{q\}$$

es decir que H incluye como axiomas a todas las aserciones de Tr. El agregado del conjunto Tr a H es necesario para la completitud del método, como veremos enseguida.

El siguiente es un primer ejemplo muy simple de prueba en H. Volvemos al programa S_{swap} que intercambia el contenido de dos variables, probado en la clase pasada utilizando directamente la definición semántica de PLW (ver Ejemplo 11.1).

Ejemplo 12.1. Prueba en H del programa S_{swap}

Dado S_{swap} :: z := x ; x := y ; y := z, se va a probar

$$\{x = X \wedge y = Y\}\ S_{swap}\ \{y = X \wedge x = Y\}$$

Por la forma del programa, recurrimos al axioma ASI tres veces, una por cada asignación, y al final completamos la prueba utilizando la (generalización de la) regla SEC:

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)

Fin de Ejemplo

Notar cómo el axioma ASI impone una forma de prueba de la postcondición a la precondición. Por la sensatez de H, entonces se cumple

$$\models \{x = X \wedge y = Y\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}$$

Obviamente también se cumple la misma fórmula pero permutando los operandos de la precondición, es decir

$$\models \{y = Y \wedge x = X\}\ z := x\ ;\ x := y\ ;\ y := z\ \{y = X \wedge x = Y\}$$

Para probar esta última fórmula tenemos que recurrir a la regla CONS, agregándole a la prueba anterior los siguientes dos pasos:

5. $(y = Y \wedge x = X) \rightarrow (x = X \wedge y = Y)$ (MAT)

6. $\{y = Y \wedge x = X\}$ z := x ; x := y ; y := z $\{y = X \wedge x = Y\}$ (4, 5, CONS)

La aserción del paso 5 pertenece al conjunto Tr, por eso se justifica el paso con el indicador MAT (por matemáticas). Notar que si H no tuviera la regla CONS, fórmulas tan simples como la del paso 6 no podrían ser probadas (H sería incompleto). Siguiendo con la fórmula del paso 6, al estar las variables de especificación X e Y implícitamente cuantificadas universalmente, se establece entonces que el programa S_{swap} se comporta adecuadamente cualesquiera sean los valores iniciales de las variables x e y. Para instanciar una fórmula de esta naturaleza con valores específicos, por ejemplo los que utilizamos en el Ejemplo 11.1, es decir

$$\{y = 2 \wedge x = 1\}\ S_{swap}\ \{y = 1 \wedge x = 2\}$$

se necesita introducir una nueva regla:

Regla de instanciación (INST) $\dfrac{f(X)}{f(c)}$

tal que f es una fórmula de correctitud que incluye una variable de especificación X, y c está en el dominio de X. La razón de utilizar INST a pesar de contar con todas las aserciones de Tr como axiomas (e implícitamente con el cálculo de predicados como mecanismo para producir aserciones verdaderas en H), es que se deben instanciar fórmulas de correctitud, no aserciones. INST es una regla universal, se incluye en todos los métodos de verificación (en la Clase 15 la utilizamos en la verificación de programas con procedimientos).

En el ejemplo siguiente se muestra el uso de la regla COND.

Ejemplo 12.2. Prueba en H de un programa que calcula el valor absoluto

El siguiente programa calcula en una variable *y* el valor absoluto de una variable x:

S_{va} :: if x > 0 then y := x else y := – x fi

Probamos a continuación {true} S_{va} {$y \geq 0$}. Considerando las dos asignaciones del programa, los primeros pasos de la prueba son

1. $\{x \geq 0\}\ y := x\ \{y \geq 0\}$ (ASI)
2. $\{-x \geq 0\}\ y := -x\ \{y \geq 0\}$ (ASI)

Para poder aplicar COND, se necesita contar con un par de fórmulas tales que sus precondiciones tengan, una la forma $p \wedge B$, y la otra la forma $p \wedge \neg B$. Haciendo p = true, y B = $x > 0$, la prueba se puede completar de la siguiente manera:

3. $(\text{true} \wedge x > 0) \rightarrow x \geq 0$ (MAT)
4. $(\text{true} \wedge \neg(x > 0)) \rightarrow -x \geq 0$ (MAT)
5. $\{\text{true} \wedge x > 0\}\ y := x\ \{y \geq 0\}$ (1, 3, CONS)
6. $\{\text{true} \wedge \neg(x > 0)\}\ y := -x\ \{y \geq 0\}$ (2, 4, CONS)
7. {true} if $x > 0$ then $y := x$ else $y := -x$ fi $\{y \geq 0\}$ (5, 6, COND)

Fin de Ejemplo

Obviamente, (true, $y \geq 0$) no es una especificación correcta de un programa que calcula el valor absoluto. Por ejemplo, S :: x:= 0; y := 1 satisface (true, $y \geq 0$). Una especificación correcta es ($x = X$, $y = |X|$). Queda como ejercicio verificar S_{va} con respecto a esta especificación.

Los últimos dos ejemplos que presentamos a continuación incluyen el uso de la regla REP. Volveremos a ellos en la clase siguiente para la prueba de terminación. Para acortar las pruebas, agrupamos varios pasos en uno, y sin explicitar la aserción de Tr utilizada en la aplicación de la regla CONS.

Ejemplo 12.3. Prueba en H de los programas del factorial y la división entera

Ya presentamos el programa S_{fac} para calcular el factorial en la clase pasada. Vamos a probar

$$\{x > 0\}\ S_{fac} :: a := 1\ ;\ y := 1\ ;\ \text{while } a < x \text{ do } a := a + 1\ ;\ y := y\ .\ a \text{ od } \{y = x!\}$$

Se propone como invariante del while la aserción p = (y = a! $\wedge$ a $\leq$ x). Notar que el invariante es muy similar a la postcondición. En lugar de x, p utiliza una variable *a* para generalizar la postcondición. Cuando el programa termina se cumple la condición a = x, y así se alcanza la postcondición buscada. La prueba se estructura de la siguiente manera:

a. {x > 0} a := 1 ; y := 1 {y = a! $\wedge$ a $\leq$ x}
b. {y = a! $\wedge$ a $\leq$ x} while a < x do a := a + 1 ; y := y . a od {y = x!}
c. {x > 0} S_{fac} {y = x!}, por la aplicación de 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. {x > 0} a := 1 ; y := 1 {y = a! $\wedge$ a $\leq$ x} (1, 2, SEC, CONS)

Prueba de b.

4. {y . a = a! $\wedge$ a $\leq$ x} y := y . a {y = a! $\wedge$ a $\leq$ x} (ASI)
5. {y . (a + 1) = (a + 1)! $\wedge$ (a + 1) $\leq$ x} a := a + 1 {y . a = a! $\wedge$ a $\leq$ x} (ASI)
6. {y = a! $\wedge$ a $\leq$ x $\wedge$ a < x } a := a + 1 ; y := y . a {y = a! $\wedge$ a $\leq$ x} (4, 5, SEC, CONS)
7. {y = a! $\wedge$ a $\leq$ x} while a < x do a := a + 1 ; y := y . a od {y = x!} (6, REP, CONS)

Prueba de c.

8. {x > 0} S_{fac} {y = x!} (3, 7, SEC)

El segundo ejemplo considera un programa que efectúa la división entera entre dos números naturales x e y, por el método de restas sucesivas. El programa obtiene el cociente en una variable c y el resto de la división en una variable r:

S_{div} :: c := 0 ; r := x ; while r $\geq$ y do r := r – y ; c := c + 1 od

Históricamente, este programa fue el primero que se probó utilizando el método H. Se quiere probar

$\{x \geq 0 \wedge y \geq 0\}\ S_{div}\ \{x = c.y + r \wedge 0 \leq r < y\}$

Notar que S_{div} no termina cuando el valor de y es cero. Se propone como invariante del while la aserción $p = (x = c.y + r \wedge 0 \leq r)$. Como en el ejemplo anterior, el invariante es muy similar a la postcondición. Cuando el programa termina se cumple $r < y$, y así se alcanza la postcondición buscada. La prueba se puede estructurar de la siguiente manera:

a. $\{x \geq 0 \wedge y \geq 0\}\ c := 0\ ;\ r := x\ \{p\}$. Las asignaciones iniciales conducen al cumplimiento por primera vez del invariante p.
b. $\{p \wedge r \geq y\}\ r := r - y\ ;\ c := c + 1\ \{p\}$. El invariante p se preserva al terminar toda iteración de la instrucción while.
c. $(p \wedge \neg(r \geq y)) \rightarrow x = c.y + r \wedge 0 \leq r < y$. Cuando finaliza el while, se cumple la postcondición del programa (implicada por la conjunción del invariante p y la negación de la condición del while).

La prueba completa no presenta mayores dificultades y queda como ejercicio.

Fin de Ejemplo

Por la completitud del método H, agregarle axiomas y reglas es redundante. De todos modos, como en cualquier sistema deductivo, esta práctica es usual a los efectos de facilitar las pruebas. Los siguientes son algunos ejemplos de axiomas y reglas auxiliares bastante comunes que se le agregan a H:

Regla de la disyunción (OR) $\dfrac{\{p\}\ S\ \{q\}\ ,\ \{r\}\ S\ \{q\}}{\{p \vee r\}\ S\ \{q\}}$

Regla de la conjunción (AND) $\dfrac{\{p_1\}\ S\ \{q_1\}\ ,\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$

Axioma de invariancia (INV) $\{p\}\ S\ \{p\}$, cuando $free(p) \cap change(S) = \varnothing$

La regla OR es útil para probar una fórmula de correctitud cuya precondición es la disyunción de dos aserciones (se puede generalizar a más aserciones). En lugar de propagar a lo largo de una prueba información necesaria para establecer oportunamente

la disyunción, se puede recurrir a esta regla, que permite una prueba por casos. Una forma particular de la regla OR de uso habitual es

$$\frac{\{p \wedge r\}\ S\ \{q\}\ ,\ \{p \wedge \neg r\}\ S\ \{q\}}{\{p\}\ S\ \{q\}}$$

Esta regla es útil cuando la prueba de {p} S {q} se facilita reforzando la precondición con dos aserciones complementarias.

La regla AND sirve para probar una fórmula de correctitud tal que su precondición y su postcondición son conjunciones de dos aserciones (se puede generalizar a más aserciones). Se recurre a esta regla para lograr una prueba incremental, y así evitar propagar información necesaria para establecer oportunamente las conjunciones.

El axioma INV suele emplearse en combinación con la regla AND. De hecho se suele utilizar directamente la siguiente regla, denominada justamente regla de invariancia:

$$\frac{\{p\}\ S\ \{q\}}{\{r \wedge p\}\ S\ \{r \wedge q\}} \text{ cuando } free(r) \cap change(S) = \varnothing$$

Existe una presentación alternativa de las pruebas en el método H, las *proof outlines* (esquemas de prueba). En una *proof outline* se intercalan los distintos pasos de la prueba de un programa entre sus instrucciones. De esta manera se obtiene una prueba mucho más estructurada, que documenta adecuadamente el programa. Como se verá después, en la verificación de los programas concurrentes las *proof outlines* son imprescindibles.

Dada la fórmula de correctitud {p} S {q}, la idea es anotar S con aserciones antes y después de cada uno de sus subprogramas S', digamos pre(S') y post(S'), respectivamente, provenientes de una prueba en el método H. La definición inductiva de una *proof outline* de {p} S {q} es la siguiente:

1. $p \rightarrow pre(S),\ post(S) \rightarrow q$
2. Si S' :: skip, entonces $pre(S') \rightarrow post(S')$
3. Si S' :: x := e, entonces $pre(S') \rightarrow post(S')[x \mid e]$
4. Si S' :: S_1 ; S_2, entonces $pre(S') \rightarrow pre(S_1)$, $post(S_1) \rightarrow pre(S_2)$, $post(S_2) \rightarrow post(S')$

5. Si S' :: if B then S_1 else S_2 fi, entonces $pre(S') \wedge B \rightarrow pre(S_1)$, $pre(S') \wedge \neg B \rightarrow pre(S_2)$, $post(S_1) \rightarrow post(S')$, $post(S_2) \rightarrow post(S')$
6. Si S' :: while B do S_1 od, entonces $pre(S') \wedge B \rightarrow pre(S_1)$, $pre(S') \wedge \neg B \rightarrow post(S')$, $post(S_1) \rightarrow pre(S')$

Por ejemplo, la siguiente es una *proof outline* correspondiente al programa del factorial que probamos en el Ejemplo 12.3 (notar que no siempre hay aserciones entre dos instrucciones consecutivas; se dice en este caso que la *proof outline* es *no estándar*):

$\{x > 0\}$
a := 1 ; y := 1 ;
$\{y = a! \wedge a \leq x\}$
while a < x do
$\{y = a! \wedge a < x\}$
a := a + 1 ; y := y . a
$\{y = a! \wedge a \leq x\}$
od
$\{y = x!\}$

Es común presentar *proof outlines* sólo con las aserciones más importantes. Mínimamente deben incluir la precondición, los invariantes de los while y la postcondición.

Ejercicios de la Clase 12

1. Considerando los axiomas y reglas de inferencia que sistematizan la aritmética, presentados en el Ejercicio 11 de la Clase 4, se pide probar el teorema 1 + 1 = 2.
2. Probar que de la regla SEC de H se puede derivar la siguiente generalización:

$$\frac{\{p\}\ S_1\ \{r_1\}\ ,\ \{r_1\}\ S_2\ \{r_2\}\ ,\ \ldots\ ,\ \{r_{n-1}\}\ S_n\ \{q\}}{\{p\}\ S_1\ ;\ S_2\ ;\ \ldots\ ;\ S_n\ \{q\}}$$

3. Extender el método H con:
   i. Un axioma para la asignación paralela $(x_1, x_2) := (e_1, e_2)$.

ii. Una regla para la instrucción if B then S fi (considerada en el Ejercicio 4 de la Clase 11).

iii. Una regla para la instrucción repeat S until B (también considerada en el Ejercicio 4 de la Clase 11).

4. Determinar si se pueden probar en H las siguientes fórmulas de correctitud parcial:
   i. {true} x := 0 {false}
   ii. {x = 0} while z = 0 do z := 0 od {x = 1}
   iii. {p} x := e {p $\wedge$ x = e}
5. Probar en H las siguientes fórmulas de correctitud parcial:
   i. {true} S {true}, para todo S.
   ii. {false} S {q}, para todo S y q.
   iii. {p} S {p}, para todo S y p tal que free(p) $\cap$ change(S) = $\varnothing$.
   iv. {p} while p do S od {q}, asumiendo que {p $\vee$ q} S {p $\vee$ q} se prueba en H.
6. Probar en H la fórmula {x = X} S_{abs} {y = |X|}, siendo S_{abs} el programa del valor absoluto mostrado en la Clase 12.
7. Desarrollar la prueba del Ejemplo 12.3, de correctitud parcial del programa S_{idiv} de la división entera, con respecto a la especificación ($x \geq 0 \wedge y \geq 0$, $x = c.y + r \wedge 0 \leq r < y$). Probar también la correctitud parcial de S_{idiv} :
   i. Con respecto a la precondición $x \geq 0 \wedge y > 0$ y la misma postcondición que antes.
   ii. Con respecto a la especificación ($x \geq 0 \wedge y = 0$, false).
8. Probar en H la fórmula $\{x \geq 0 \wedge y \geq 0\}$ S_{prod} {prod = x . y}, siendo S_{prod} un programa que calcula el producto entre x e y de la siguiente manera:
   S_{prod} :: prod := 0 ; k := y ; while k > 0 do prod := prod + x ; k := k – 1 od.
9. Probar en H la correctitud parcial de S :: while x < y do x := x + 1 ; y := y – 1 od, con respecto a la especificación ($x = X \wedge y = Y \wedge X < Y$, $[par(X + Y) \rightarrow x = y = (X + Y) / 2] \wedge [impar(X + Y) \rightarrow x = (X + Y + 1) / 2 \wedge y = (X + Y – 1) / 2]$). Se permite utilizar la regla AND, y la siguiente regla denominada SHIFT:

$$\frac{\{p \wedge q\}\ S\ \{r\},\ free(q) \cap change(S) = \varnothing}{\{p\}\ S\ \{q \rightarrow r\}}$$

10. Probar en H la fórmula $\{x = X \wedge y = Y \geq 0\}$ S $\{z = X^Y\}$, siendo S el programa:
    S :: z := 1; while $y \neq 0$ do if par(y) then y := y/2; $x := x^2$ else y := y - 1; z := z.x fi od.

Clase 13. Verificación de la terminación

En esta clase completamos la presentación de la metodología de verificación describiendo el método H*, que permite probar la terminación de los programas PLW.
Una prueba en H* de la terminación de un programa S a partir de una precondición p, se denota con la expresión $\vdash_{H^*} \langle p \rangle\ S\ \langle true \rangle$. Por la sensatez del método, $\vdash_{H^*} \langle p \rangle\ S\ \langle true \rangle$ implica $\models \langle p \rangle\ S\ \langle true \rangle$, es decir que a partir de todo estado σ que satisface p, S termina.
Los métodos H y H* difieren sólo en la regla de la repetición REP, porque el while es la única instrucción que puede provocar no terminación. Como convención de notación, a los nombres de los axiomas y reglas de H* se les agrega al final el símbolo *, y las pre y postcondiciones se delimitan con ⟨ ⟩.
La única novedad a presentar es, entonces, la regla REP*. REP* mantiene la idea de un invariante p, pero ahora parametrizado con un *variante* n, que es una variable libre en p, no es una variable de programa, y su dominio son los números naturales. La regla relaciona todo while con una variable de estas características que se decrementa con cada iteración, lo que permite asegurar que la cantidad de iteraciones sea finita.
Presentamos a continuación los componentes de H*.

Definición 13.1. Axiomas y reglas del método H*

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

$$\text{Regla REP*} \qquad \frac{p(n+1) \rightarrow B\ ,\ \langle p(n+1) \rangle\ S\ \langle p(n) \rangle\ ,\ p(0) \rightarrow \neg B}{\langle \exists n\colon p(n) \rangle\ \text{while } B \text{ do } S \text{ od } \langle p(0) \rangle}$$

Fin de Definición

El invariante parametrizado p(n) en realidad abrevia nat(n) $\wedge$ p(n), siendo nat un predicado unario que caracteriza a los números naturales. Al igual que las variables de especificación, el variante n en las premisas de REP* debe entenderse como implícitamente cuantificado universalmente. Como en la regla REP de H, REP*

requiere que la ejecución de S mantenga invariante p mientras se cumpla B. Y ahora se requiere además lo siguiente:

- Si n no es cero, entonces B debe ser verdadera (notar que n + 1 es al menos uno).
- Con la ejecución de S, n debe decrementarse en uno.
- Cuando n llega a cero, B debe ser falsa.

Con estas premisas se llega a la conclusión ⟨∃n: p(n)⟩ while B do S od ⟨p(0)⟩, es decir que el while termina (al cabo de n iteraciones, para algún n) a partir de un estado que satisface un invariante p (que entonces sigue valiendo después del while).
Notar que en H* podemos demostrar directamente la correctitud total de un programa S con respecto a una especificación (p, q): si los invariantes de los while de S propagan la información necesaria para alcanzar al final de la prueba la postcondición q, no hace falta partir la prueba de ⟨p⟩ S ⟨q⟩ en las pruebas de {p} S {q} y ⟨p⟩ S ⟨true⟩ como se formula en el Lema de Separación. No obstante, suele ser más sencillo desarrollar dos pruebas, las cuales son de naturaleza distinta. Es más, cuando hay varias propiedades para probar como en el caso de los programas concurrentes, lo habitual es desarrollar una prueba para cada propiedad.
En lo que sigue probamos la terminación del programa del factorial, cuya correctitud parcial se demostró en el Ejemplo 12.3.

Ejemplo 13.1. Prueba en H* de terminación del programa del factorial

Ya hemos verificado en H que

$$\{x > 0\}\ S_{fac} :: a := 1 ; y := 1 ; \text{while } a < x \text{ do } a := a + 1 ; y := y . a \text{ od } \{y = x!\}$$

Ahora probaremos en H* que

$$\langle x > 0 \rangle\ S_{fac}\ \langle true \rangle$$

Se propone como invariante parametrizado de la repetición, la aserción

$$p(n) = (n = x - a)$$

Informalmente, la utilidad de p(n) se justifica de esta manera. Al comienzo del while se cumple $x > 0 \wedge a = 1$, y así, x – a arranca con un valor natural. Luego, x – a decrece en uno con cada iteración, por la asignación a := a + 1, y cuando llega a cero deja de valer la condición a < x. Formalmente, vamos a probar

a. $\langle x > 0 \rangle$ a := 1 ; y := 1 $\langle \exists n: n = x - a \rangle$
b. $\langle \exists n: n = x - a \rangle$ while a < x do a := a + 1 ; y := y . a od $\langle true \rangle$
c. $\langle x > 0 \rangle$ S_{fac} $\langle true \rangle$, por aplicación de SEC* sobre a y b

Prueba de a.

1. $\langle \exists n: n = x - 1 \rangle$ a := 1; y := 1 $\langle \exists n: n = x - a \rangle$ (ASI*, SEC*)
2. $x > 0 \rightarrow \exists n: n = x - 1$ (MAT)
3. $\langle x > 0 \rangle$ a := 1; y := 1 $\langle \exists n: n = x - a \rangle$ (1, 2, CONS*)

Prueba de b.

(Se prueban las tres premisas de REP*)

4. $n + 1 = x - a \rightarrow a < x$ (MAT)
5. $\langle n + 1 = x - a \rangle$ a := a + 1 ; y := y . a $\langle n = x - a \rangle$ (ASI*, SEC*, CONS*)
6. $0 = x - a \rightarrow \neg(a < x)$ (MAT)

(Se establece la conclusión de REP*)

7. $\langle \exists n: n = x - a \rangle$ while a < x do a := a + 1 ; y := y . a od $\langle 0 = x - a \rangle$ (4, 5, 6, REP*)
8. $0 = x - a \rightarrow true$ (MAT)
9. $\langle \exists n: n = x - a \rangle$ while a < x do a := a + 1 ; y := y . a od $\langle true \rangle$ (7, 8, CONS*)

Prueba de c.

10. $\langle x > 0 \rangle$ S_{fac} $\langle true \rangle$ (3, 9, SEC*)

Fin de Ejemplo

Existe una forma alternativa más flexible de la regla REP*, que no exige un decremento estricto en uno del variante ni tampoco que al final su valor sea cero. También se puede considerar un orden parcial bien fundado distinto de (N, <). De esta manera se pueden

facilitar las pruebas en H*, pero como contrapartida se dificulta la demostración de la sensatez y completitud del método.

Por ejemplo, para probar la terminación del siguiente programa, que calcula el máximo común divisor de dos números naturales x_1 y x_2, denotado por $mcd(x_1, x_2)$ (ya nos hemos referido a esta función en el Ejemplo 7.2), es más sencillo emplear la forma alternativa de REP*:

S_{mcd} :: $y_1 := x_1$; $y_2 := x_2$;
while $y_1 \neq y_2$ do
if $y_1 > y_2$ then $y_1 := y_1 - y_2$ else $y_2 := y_2 - y_1$ fi
od

Se cumple $\langle x_1 > 0 \wedge x_2 > 0 \rangle\ S_{mcd}\ \langle y_1 = mcd(x_1, x_2) \rangle$. Para la prueba de correctitud parcial se puede utilizar la relación invariante $mcd(x_1, x_2) = mcd(y_1, y_2)$, y para la prueba de terminación, el variante

$$n = y_1 + y_2$$

En este caso, n no decrece de a uno ni termina en cero. Otra posibilidad es recurrir al orden parcial bien fundado $(N \times N, <)$, con la relación menor habitual $(n_1, n_2) < (m_1, m_2) \leftrightarrow (n_1 < m_1 \vee (n_1 = m_1 \wedge n_2 < m_2))$. En este caso, un variante posible es

$$(n_1, n_2) = (y_1, y_2)$$

Después de cada iteración, la variable y_1 o la variable y_2 se decrementan y siguen siendo mayores que cero. No se alcanza nunca el valor minimal (0, 0).

Otra regla habitual para la prueba de terminación de los while utiliza, en lugar de un invariante parametrizado p(n), una expresión entera t, denominada *función cota*, que se define en términos de las variables de programa. La regla es la siguiente (para distinguirla de la anterior la denominamos REP**):

Regla REP** $$\frac{\langle p \wedge B \rangle\ S\ \langle p \rangle\ ,\ \langle p \wedge B \wedge t = Z \rangle\ S\ \langle t < Z \rangle\ ,\ p \rightarrow t \geq 0}{\langle p \rangle \text{ while } B \text{ do } S \text{ od } \langle p \wedge \neg B \rangle}$$

La variable Z es una variable de especificación, no aparece en p, B, t ni S, y su objetivo es conservar el valor de la expresión t antes de la ejecución del cuerpo del while. La primera premisa es la misma que la de la regla REP de H. Las otras dos premisas son las que aseguran la terminación:

- Por la segunda premisa, la expresión t se decrementa en cada iteración.
- Por la tercera premisa, t arranca y se mantiene no negativa después de cada iteración.

De esta manera, el while debe terminar (en un estado que satisface el invariante p).
A continuación empleamos REP** para probar la terminación del programa de la división entera por el método de restas sucesivas, cuya correctitud parcial demostramos en el Ejemplo 12.3.

Ejemplo 13.2. Prueba en H* de terminación del programa de la división entera

Ya se verificó en H que

$$\{x \geq 0 \wedge y \geq 0\}$$
$$S_{div} :: c := 0 ; r := x ; \text{while } r \geq y \text{ do } r := r - y ; c := c + 1 \text{ od}$$
$$\{x = c.y + r \wedge 0 \leq r < y\}$$

Ahora probamos la terminación de S_{div} empleando REP**. El programa no termina cuando y = 0, así que consideramos la precondición $x \geq 0 \wedge y > 0$. Vamos a aprovechar también este ejemplo para mostrar una prueba de correctitud total sin recurrir al Lema de Separación. Verificamos directamente

$$\langle x \geq 0 \wedge y > 0 \rangle \; S_{div} \; \langle x = c.y + r \wedge 0 \leq r < y \rangle$$

Se propone como invariante del while la aserción

$$p = (x = c.y + r \wedge r \geq 0 \wedge y > 0)$$

y como función cota

$t = r$

Notar que r no se decrementa en uno sino en el valor de y (que guarda el divisor) luego de cada iteración, y que su valor final no es necesariamente cero sino algún número natural menor que y. La prueba es la siguiente (para simplificar la escritura utilizamos el identificador p del invariante en lugar de la aserción completa):

1. $\langle x \geq 0 \wedge y > 0 \rangle$ c := 0 ; r := x $\langle p \rangle$ (ASI*, SEC*, CONS*)

(Se prueban las tres premisas de REP**)

2. $\langle p \wedge r \geq y \rangle$ r := r – y ; c := c + 1 $\langle p \rangle$ (ASI*, SEC*, CONS*)

3. $\langle p \wedge r \geq y \wedge r = Z \rangle$ r := r – y ; c := c + 1 $\langle r < Z \rangle$ (ASI*, SEC*, CONS*)

4. $p \rightarrow r \geq 0$ (MAT)

(Se establece la conclusión de REP**)

5. $\langle p \rangle$ while $r \geq y$ do r := r – y ; c := c + 1 od $\langle p \wedge \neg(r \geq y) \rangle$ (2, 3, 4, REP**)

6. $\langle x \geq 0 \wedge y > 0 \rangle$ S_{div} $\langle x = c.y + r \wedge 0 \leq r < y \rangle$ (1, 5, SEC*, CONS*)

Fin de Ejemplo

Para probar la terminación de S_{div} con la regla REP* se puede utilizar, por ejemplo, el invariante parametrizado

$p(n) = (x = c.y + r \wedge r \geq 0 \wedge n.y \leq r < (n + 1).y)$

Queda como ejercicio desarrollar la prueba de esta manera. Consideremos ahora este último ejemplo:

S_{eps} :: while x > epsilon do x := x / 2 od

Apartándonos por un momento del dominio de los números enteros, supongamos que las variables x y epsilon del programa S_{eps} son de tipo real, inicialmente mayores que cero. Claramente se cumple $\langle x = X \rangle\ S_{eps}\ \langle true \rangle$, porque los distintos valores positivos de x constituyen iteración tras iteración una secuencia decreciente estricta, y epsilon > 0. También en este caso podemos emplear la regla REP**, definiendo una adecuada función cota en el dominio de los números naturales. Una posible función es la siguiente:

if x > epsilon then $\lceil \log_2 X/epsilon \rceil - \log_2 X/x$ else 0 fi

En la clase que viene hacemos referencia a la prueba de terminación de programas considerando cualquier interpretación, y a la expresividad del lenguaje de especificación para definir funciones cota, invariantes (parametrizados o no), etc.

Las *proof outlines* de $\langle p \rangle\ S\ \langle q \rangle$ se definen de la misma manera que las *proof outlines* de {p} S {q}, agregándoles las funciones cota antes de cada while, y utilizando los delimitadores $\langle\ \rangle$ en lugar de {} (no se suele incluir la prueba de terminación, con el agregado de la función cota alcanza). Por ejemplo, en el caso del programa de la división entera, una *proof outline* no estándar de $\langle x \geq 0 \wedge y > 0 \rangle\ S_{div}\ \langle x = c.y + r \wedge 0 \leq r < y \rangle$ es

$\langle x \geq 0 \wedge y > 0 \rangle$
c := 0 ; r := x ;
$\langle$inv: $x = c.y + r \wedge r \geq 0 \wedge y > 0$, fc: r$\rangle$
while $r \geq y$ do
$\langle x = c.y + r \wedge r \geq 0 \wedge y > 0 \wedge r \geq y \rangle$
r := r – y ; c := c + 1
$\langle x = c.y + r \wedge r \geq 0 \wedge y > 0 \rangle$
od
$\langle x = c.y + r \wedge 0 \leq r < y \rangle$

Como se muestra en el ejemplo, se suele especificar antes de cada while el invariante precedido por el identificador inv, y la función cota precedida por el identificador fc.

El *fairness* es un concepto íntimamente relacionado con la terminación de los programas, en realidad con cualquier propiedad de tipo *liveness*. Establece restricciones en cuanto a qué tipo de computaciones infinitas pueden ocurrir. Los lenguajes de programación no determinísticos y concurrentes incluyen distintos tipos de *fairness*. Si bien ambos paradigmas están fuera del alcance de este libro, presentamos a continuación los aspectos más salientes del impacto del *fairness* en la terminación dada su importancia en la verificación de programas.

Consideraremos solamente una extensión no determinística de PLW, reemplazando la instrucción de repetición determinística por la siguiente variante no determinística:

$$\text{do } B_1 \to S_1 \; [] \; \ldots \; [] \; B_n \to S_n \text{ od}$$

Al momento de la ejecución de la repetición, se evalúan todas las condiciones booleanas B_i, de las que resultan verdaderas se elige no determinísticamente una (si todas son falsas la repetición termina), se ejecuta la instrucción asociada S_i, y se vuelve al comienzo. Las instrucciones $B_i \to S_i$ se conocen como *comandos con guardia*, y las condiciones B_i se denominan *guardias booleanas* o directamente *guardias*. Las i son las *direcciones* de la repetición. Cuando una guardia B_i es verdadera, se dice que la dirección i está *habilitada*.

Por ejemplo, el siguiente programa S_{num} devuelve algún número natural entre 0 y K, a partir de una precondición $x = 0 \wedge K \geq 0 \wedge b$ (para facilitar la escritura permitimos el uso de variables booleanas como b):

S_{num} :: do
    1: $b \wedge x < K \to x := x + 1$
    []
    2: $b \to b := \text{false}$
    od

Los números 1 y 2 se utilizan para identificar las dos direcciones de S_{num}, no son parte de la sintaxis. Claramente el programa termina, y no hay necesidad de ninguna hipótesis de *fairness*.

La siguiente es una regla habitual para la prueba de terminación en este caso:

INI. $p \rightarrow \exists n: p(n)$

CONT. $p(n+1) \rightarrow \bigvee_{i=1,n} B_i$

DEC. $\langle p(n) \wedge n > 0 \wedge B_i \rangle\ S_i\ \langle \exists k: k < n \wedge p(k) \rangle$, para todo i

TERM. $p(0) \rightarrow \bigwedge_{i=1,n} \neg B_i$

$\langle p \rangle$ do $B_1 \rightarrow S_1 \square \ldots \square B_n \rightarrow S_n$ od $\langle p(0) \rangle$

Como se aprecia, la regla es una adaptación no determinística de la regla REP* de H*. Las premisas INI, CONT, DEC y TERM establecen las condiciones de inicio, continuidad, decremento y terminación de la repetición, respectivamente. Se mantiene la idea de un invariante parametrizado p(n). La ejecución de todo S_i debe preservar p cuando se ejecuta a partir de la condición $p \wedge B_i$. Notar que en cada dirección, el decremento del variante n puede ser distinto (n representa de esta manera la máxima cantidad posible de iteraciones).

Para probar $\langle x = 0 \wedge K \geq 0 \wedge b \rangle\ S_{num}\ \langle true \rangle$ se puede utilizar el invariante parametrizado

$$p(n) = (b \wedge x < K \rightarrow n = K - x + 1) \wedge (b \wedge x \geq K \rightarrow n = 1) \wedge (\neg b \rightarrow n = 0)$$

El decremento de n tomando la dirección 1 es uno, y tomando la dirección 2 es variable, depende de su valor actual, porque incondicionalmente pasa a valer cero. Queda como ejercicio desarrollar la prueba.

Si en cambio se quiere que el programa S_{num} devuelva cualquier número natural, se lo debe modificar por ejemplo de la siguiente manera:

S_{num2} :: do

1: $b \rightarrow x := x + 1$

$\square$

2: $b \rightarrow b :=$ false

od

Pero ahora, sin ninguna hipótesis de *fairness*, este programa puede no terminar, eligiendo siempre la dirección 1. Si en cambio se asegura que una dirección habilitada permanentemente a partir de un momento dado no puede ser postergada

indefinidamente, S_{num} termina siempre, porque alguna vez va a elegir la dirección 2. Dicha hipótesis se conoce como *fairness débil* (mientras no introduzcamos una hipótesis distinta, toda mención al *fairness* en lo que sigue debe entenderse como *fairness* débil). En términos del "árbol de computaciones" de un programa no determinístico (dado que ahora un programa puede tener varias computaciones y así varios estados finales), el *fairness* implica la "poda" de las computaciones inválidas o no *fair* (no justas), que son las computaciones infinitas con direcciones habilitadas permanentemente a partir de un momento dado que no se eligen nunca.

Con hipótesis de *fairness*, la verificación de la terminación se puede relajar: no hace falta que el variante se decremente cualquiera sea la dirección elegida, sino que alcanza con que lo haga en determinadas direcciones *útiles*. Una regla habitual para la prueba de terminación con *fairness* es la siguiente. Dada una instrucción de repetición do $B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n$ od, si

- (W, <) es un orden parcial bien fundado, con un minimal w_0
- p(w) es un invariante parametrizado
- Para todo $w \in W$, con $w > w_0$, $\Delta_w = \Delta^d_w \cup \Delta^s_w$ es una partición del conjunto Δ de direcciones, siendo $\Delta^d_w \neq \varnothing$
- Se satisfacen las siguientes premisas:

  INI. $p \to \exists w: p(w)$

  CONT. $(\forall w > w_0)(\exists i \in \Delta^d_w): p(w) \to B_i$

  DEC. $\langle p(w) \wedge w > w_0 \wedge B_i \rangle\ S_i\ \langle \exists v \in W: v < w \wedge p(v) \rangle$, para todo $i \in \Delta^d_w$

  NOINC. $\langle p(w) \wedge w > w_0 \wedge B_i \rangle\ S_i\ \langle \exists v \in W: v \leq w \wedge p(v) \rangle$, para todo $i \in \Delta^s_w$

  TERM. $p(w_0) \to \bigwedge_{i=1,n} \neg B_i$, con $i \in \Delta$

entonces se cumple

$\langle p \rangle$ do $B_1 \to S_1 \,[]\, \ldots \,[]\, B_n \to S_n$ od $\langle p(w_0) \rangle$

En palabras, la regla establece:

- Por cada valor w 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, es decir que no alargan la distancia a la terminación, porque 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 w_0, existe una determinada guardia B_i que es verdadera, siendo i una dirección útil. (Al haber *fairness*, la dirección i será elegida alguna vez, y entonces el variante se decrementará inexorablemente.)
- Las premisas INI y TERM son las mismas que las de la regla de terminación sin *fairness*, salvo que ahora hacen referencia a cualquier orden bien fundado, no solamente (N, <). Por INI, el invariante p asegura que existe un valor inicial para el parámetro w, que representa la distancia máxima a la terminación, y por TERM, cuando se alcanza el minimal la repetición termina.

Por ejemplo, para el programa anterior S_{num2} :: do 1: $b \to x := x + 1$ □ 2: $b \to b := false$ od, puede utilizarse el invariante parametrizado

$$p(w) = (b \to w = 1) \wedge (\neg b \to w = 0)$$

La dirección que ayuda es la 2. Queda como ejercicio desarrollar la prueba. Consideremos ahora este otro programa:

S_{num3} :: do

1: $b_1 \wedge b_2 \to x := x + 1$

□

2: $b_1 \wedge b_2 \to b_1 := false$

□

3: $\neg b_1 \wedge b_2 \to y := y + 1$

□

4: $\neg b_1 \wedge b_2 \to b_2 := false$

od

tal que inicialmente b_1 y b_2 son verdaderas, y x = y = 0. El programa S_{num3} tiene el mismo propósito que S_{num2}, pero ahora devuelve dos números naturales cualesquiera en lugar de uno. Sin *fairness*, S_{num3} puede no terminar, eligiendo al comienzo siempre la dirección 1, o bien eligiendo siempre la dirección 3 cuando b_1 es falsa y b_2 es verdadera. Con *fairness* esto no sucede. Se puede utilizar el orden (W = {0, 1, 2}, <) con la relación < habitual, las particiones $\Delta^d_2 = \{2\}$ y $\Delta^s_2 = \{1, 3, 4\}$ cuando w = 2 y $\Delta^d_1 = \{4\}$ y $\Delta^s_1 = \{1, 2, 3\}$ cuando w = 1, y el invariante parametrizado

$$p(w) = (b_1 \wedge b_2 \rightarrow w = 2) \wedge (\neg b_1 \wedge b_2 \rightarrow w = 1) \wedge$$
$$(\neg b_1 \wedge \neg b_2 \rightarrow w = 0) \wedge (b_1 \rightarrow b_2)$$

Queda como ejercicio desarrollar la prueba. Otra aproximación para verificar la terminación de los programas no determinísticos con *fairness*, se basa en el uso de las *asignaciones aleatorias*. Este tipo de instrucción introduce no determinismo a través de los datos. Tiene la forma x := ?, y su efecto es asignar a la variable x algún número natural. El axioma habitual asociado a la asignación aleatoria es $\{\forall x: p\}\ x := ?\ \{p\}$, para toda aserción p. Un ejemplo de programa con asignaciones aleatorias es el siguiente:

S_{al} :: do
$b \wedge x > 0 \rightarrow x := x - 1$
□
$b \wedge x < 0 \rightarrow x := x + 1$
□
$\neg b \rightarrow x := ? ; b := true$
od

Este programa termina sin hipótesis de *fairness*. Si al comienzo b es verdadera, S_{al} termina al cabo de |x| iteraciones, y si es falsa, no se puede predecir el número de iteraciones porque este valor se conoce recién cuando se efectúa la asignación aleatoria, pero seguro que la cantidad también es finita. Un variante n asociado a la repetición debe cumplir inicialmente, entonces, $n \geq |x|$, para todo $x \geq 0$, lo cual no es posible dentro

de (N, <). Así, se debe recurrir a otro orden parcial bien fundado, por ejemplo (N ∪ {ω}, <), siendo ω el primer ordinal infinito.

La aproximación alternativa para probar la terminación con *fairness* de un programa S, utilizando asignaciones aleatorias, consiste en implementar dentro de S un *planificador* que asegure que haya *fairness*:

- Primero se amplía S con asignaciones aleatorias, con el objetivo de anular todas las computaciones que no sean *fair*.
- Luego se verifica la terminación del programa S modificado, sin hipótesis de *fairness*, y recurriendo al axioma de asignación aleatoria.

Precisamos la idea sobre el esquema de programa S :: do 1: $B_1 \rightarrow S_1$ □ 2: $B_2 \rightarrow S_2$ od, y aprovechamos para introducir un segundo tipo de *fairness*, el *fairness fuerte*. En este caso, alcanza con que a partir de un momento dado una dirección esté habilitada infinitas veces (no importa que sea de manera intermitente) para que no pueda ser postergada indefinidamente. Notar que el *fairness* fuerte implica el *fairness* débil. La implementación del planificador en el programa S consiste en lo siguiente:

S' :: z_1 := ? ; z_2 := ? ;
do
1: $B_1 \wedge z_1 \leq z_2 \rightarrow S_1$; z_1 := ? ; if B_2 then $z_2 := z_2 - 1$ else skip fi
□
2: $B_2 \wedge z_2 < z_1 \rightarrow S_2$; z_2 := ? ; if B_1 then $z_1 := z_1 - 1$ else skip fi
od

Es decir, se introducen dos variables que no están en S, z_1 y z_2, que 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. Al comienzo, z_1 y z_2 se inicializan con valores arbitrarios no negativos. Dada z_i , 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 se cumple después de las reinicializaciones aleatorias que están dentro de la repetición. Después de ejecutarse S_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 (en algún momento el valor de z_k será negativo, y con z_i := ? la variable z_i sólo recibe valores mayores o iguales que cero). Notar que no importa que B_k sea verdadera intermitentemente, por eso esta aproximación funciona cuando hay *fairness* fuerte. Eliminadas las computaciones no *fair* de esta manera, luego se desarrolla la prueba de terminación sin hipótesis de *fairness*, mediante los axiomas y reglas presentados anteriormente, incluyendo el axioma de asignación aleatoria.

Ejercicios de la Clase 13

1. Probar que todo subconjunto no vacío W de un conjunto bien fundado tiene al menos un elemento minimal w_0, es decir un elemento w_0 tal que no existe ningún otro elemento $w \in W$ que cumple que $w < w_0$.
2. Plantear una forma más flexible para la regla REP* presentada en la Clase 13.
3. Probar la terminación de los siguientes programas:
   i. El programa S_{idiv} de la división entera mostrado en la Clase 13, con respecto a la precondición $x \geq 0 \wedge y > 0$, utilizando ahora la regla REP* (en la Clase 13 ya se probó su correctitud total utilizando la regla REP**). Se podría utilizar el invariante parametrizado $p(n) = (x = c.y + r \wedge r \geq 0 \wedge n.y \leq r < (n + 1).y)$, como se indicó en la misma clase.
   ii. Los programas de los Ejercicios 8, 9 y 10 de la Clase 12, considerando las mismas precondiciones.
4. Probar la correctitud total de los siguientes programas:
   i. El programa S_{mcd} del máximo común divisor mostrado en la Clase 13, con respecto a la especificación ($x_1 > 0 \wedge x_2 > 0$, $y_1 = mcd(x_1, x_2)$). Se podría considerar la relación invariante $mcd(x_1, x_2) = mcd(y_1, y_2)$, y el variante $n = y_1 + y_2$, como se indicó en la misma clase.
   ii. $\langle \exists z \geq 0: x = X = 2^z \wedge y = 0 \rangle$
   while par(x) do x := x / 2 ; y := y + 1 od
   $\langle X = 2^y \rangle$
   iii. $\langle \exists k: k > 1 \wedge y = 3k \rangle$
   x := 0 ; while x < y do x := x + 1 ; y := y – 2 od
   $\langle y = x \rangle$

5. Probar que el *fairness* fuerte implica el *fairness* débil.
6. Determinar con qué hipótesis de *fairness* terminan los siguientes programas no determinísticos de PLW, asumiendo la precondición true:
   i. do $b \to i := i + 1 \; [] \; b \to i := 0 \; [] \; b \wedge i = 1 \to b := false$ od
   ii. do $b \to i := i + 1 \; [] \; b \to i := 0 \; [] \; b \wedge i = 2 \to b := false$ od
7. Probar la terminación de los siguientes programas no determinísticos de PLW, asumiendo hipótesis de *fairness* débil:
   i. El programa S_{num2} mostrado en la Clase 13, a partir de la precondición $b \wedge x = 0$. Se podría utilizar el invariante parametrizado $p(w) = (b \to w = 1) \wedge (\neg b \to w = 0)$, como se indicó en la misma clase.
   ii. El programa S_{num3} mostrado en la Clase 13, a partir de la precondición $b_1 \wedge b_2 \wedge x = 0 \wedge y = 0$. Se podría utilizar el invariante parametrizado $p(w) = (b_1 \wedge b_2 \to w = 2) \wedge (\neg b_1 \wedge b_2 \to w = 1) \wedge (\neg b_1 \wedge \neg b_2 \to w = 0) \wedge (b_1 \to b_2)$, como se indicó en la misma clase.
   iii. S :: do $x = 0 \to y := y + 1 \; [] \; x = 0 \to x := 1 \; [] \; x > 0 \wedge y > 0 \to y := y - 1$ od, a partir de la precondición true.
   iv. S :: do $xup \to x := x + 1 \; [] \; yup \to y := y + 1$
   $[] \; xup \to xup := false \; [] \; yup \to yup := false$
   $[] \; x > 0 \to x := x - 1 \; [] \; y > 0 \to y := y - 1$ od,
   a partir de la precondición true.

Clase 14. Sensatez y completitud de los métodos de verificación

Presentados los métodos H y H* para las pruebas de correctitud parcial y terminación de programas, respectivamente, en esta clase demostramos su sensatez y completitud. Dado el carácter introductorio de este libro, hemos optado por presentar los conceptos básicos de la verificación de programas fijando un lenguaje de programación, PLW, y un lenguaje de especificación, Assn, considerando la interpretación estándar de los números enteros. En este contexto probamos en lo que sigue la sensatez y la completitud de H y H*. Luego analizamos ambas propiedades en un marco más general.
Propio de los sistemas deductivos de la lógica, la prueba de la sensatez de un método axiomático de verificación de programas consiste en demostrar que sus axiomas son verdaderos y que sus reglas de inferencia son sensatas, es decir que a partir de premisas verdaderas obtienen conclusiones verdaderas, o en otras palabras, que preservan la verdad.
Primero probamos la sensatez del método H, en realidad de H $\cup$ Tr, siendo Tr el conjunto de todas las aserciones verdaderas sobre los números enteros considerando la interpretación estándar, extensión necesaria para la completitud de H como vimos en la Clase 12.

Teorema 14.1. Sensatez del método H

Para todo programa S de PLW y todo par de aserciones p y q de Assn, se cumple

$$\text{Tr} \vdash_{\text{H}} \{p\}\ S\ \{q\} \rightarrow \models \{p\}\ S\ \{q\}$$

Lo probaremos por inducción sobre la longitud de la prueba. Para simplificar la notación usaremos de ahora en más $\vdash$ en lugar de $\text{Tr} \vdash_{\text{H}}$ cuando el significado quede claro por contexto. En la base de la inducción consideramos los dos axiomas de H:

1. El axioma del skip (SKIP) es verdadero, es decir $\models \{p\}\ \text{skip}\ \{p\}$. Sea σ un estado inicial tal que $\sigma \models p$. Por la semántica de PLW, $(\text{skip}, \sigma) \rightarrow (E, \sigma)$. Entonces, el estado final σ cumple $\sigma \models p$.

2. El axioma de la asignación (ASI) es verdadero, es decir $\models \{p[x \mid e]\}\ x := e\ \{p\}$. Sea σ un estado inicial tal que $\sigma \models p[x \mid e]$. Por la la semántica de PLW, $(x := e, \sigma) \rightarrow (E, \sigma[x \mid e])$. Entonces, aplicando el Lema de Sustitución (ver Clase 11), el estado final $\sigma[x \mid e]$ cumple $\sigma[x \mid e] \models p$.

La prueba continúa de la siguiente manera:

3. La regla de la secuencia (SEC) es sensata, es decir que se cumple $\models \{p\}\ S_1 ; S_2\ \{q\}$ si se prueban en H las fórmulas $\{p\}\ S_1\ \{r\}$ y $\{r\}\ S_2\ \{q\}$. Supongamos que $\vdash \{p\}\ S_1\ \{r\}$ y $\vdash \{r\}\ S_2\ \{q\}$. Entonces, por hipótesis inductiva, $\models \{p\}\ S_1\ \{r\}$ y $\models \{r\}\ S_2\ \{q\}$. Sea σ_0 un estado inicial tal que $\sigma_0 \models p$ y $val(\pi(S_1 ; S_2, \sigma_0)) = \sigma_2 \neq \perp$. Por la semántica de PLW, $(S_1 ; S_2, \sigma_0) \rightarrow^* (S_2, \sigma_1) \rightarrow^* (E, \sigma_2)$. Como $\models \{p\}\ S_1\ \{r\}$, entonces $\sigma_1 \models r$. Y como $\models \{r\}\ S_2\ \{q\}$, entonces el estado final σ_2 cumple $\sigma_2 \models q$.
4. La regla del condicional (COND) es sensata, es decir que se cumple $\models \{p\}$ if B then S_1 else S_2 fi $\{q\}$ si se prueban en H las fórmulas $\{p \wedge B\}\ S_1\ \{q\}$ y $\{p \wedge \neg B\}\ S_2\ \{q\}$. La prueba es similar a la del item anterior y queda como ejercicio.
5. La regla de la repetición (REP) es sensata, es decir que se cumple $\models \{p\}$ while B do S od $\{p \wedge \neg B\}$ si se prueba en H la fórmula $\{p \wedge B\}\ S\ \{p\}$. Supongamos que $\vdash \{p \wedge B\}\ S\ \{p\}$. Entonces, por hipótesis inductiva, $\models \{p \wedge B\}\ S\ \{p\}$. Sea σ_0 un estado inicial tal que $\sigma_0 \models p$ y $val(\pi(\text{while B do S od}, \sigma_0)) \neq \perp$. Por la semántica de PLW, la computación del while a partir de σ_0 tiene la forma $C_0 \rightarrow^* \ldots \rightarrow^* C_n$, tal que $n > 0$, $C_i = (\text{while B do S od}, \sigma_i)$ con $0 \leq i < n$, y $C_n = (E, \sigma_n)$ con $\sigma_n(B) = \text{falso}$. Como $\models \{p \wedge B\}\ S\ \{p\}$, entonces $\sigma_i \models p$ con $0 \leq i \leq n$, y por lo tanto el estado final σ_n cumple $\sigma_n \models p \wedge \neg B$.
6. Finalmente, la regla de consecuencia (CONS) es sensata, es decir que se cumple $\models \{p\}\ S\ \{q\}$ si se prueba en H la fórmula $\{p_1\}\ S\ \{q_1\}$ y las aserciones $p \rightarrow p_1$ y $q_1 \rightarrow q$ pertenecen al conjunto Tr. Supongamos que $\vdash \{p_1\}\ S\ \{p_1\}$, $p \rightarrow p_1$ y $q_1 \rightarrow q$. Entonces, por hipótesis inductiva, $\models \{p_1\}\ S\ \{q_1\}$. Sea σ_0 un estado inicial tal que $\sigma_0 \models p$ y $val(\pi(S, \sigma_0)) = \sigma_1 \neq \perp$. Por la semántica de PLW, $(S, \sigma_0) \rightarrow^* (E, \sigma_1)$. Como $p \rightarrow p_1$, entonces $\sigma_0 \models p_1$. Como $\models \{p_1\}\ S\ \{q_1\}$, entonces $\sigma_1 \models q_1$. Y como $q_1 \rightarrow q$, entonces el estado final σ_1 cumple $\sigma_1 \models q$.

Fin de Teorema

De la misma manera se puede probar que los axiomas y reglas auxiliares mencionados en la Clase 12 (axioma de invariancia, regla de la disyunción, regla de la conjunción) preservan la sensatez del método H (queda como ejercicio). También se cumple la sensatez de la regla de instanciación (INST), como consecuencia directa de la definición de variable de especificación: si f(X) es una fórmula de correctitud verdadera con una variable de especificación X, entonces también lo es f(c) si c está en el dominio de X, porque X está implícitamente cuantificada universalmente. Tener en cuenta de todos modos la restricción que establecimos anteriormente, de que si una variable de especificación X aparece libre en la precondición y postcondición de una fórmula, para que pueda ser instanciada con una expresión ésta no puede incluir variables de programa.

Obviamente, la sensatez de H también se puede probar considerando las *proof outlines*, porque se cumple |– {p} S {q} si y sólo si existe una *proof outline* de {p} S {q}. Dada una *proof outline* estándar de {p} S {q}, se demuestra por inducción sobre la computación y la estructura de S que cuando la computación de S, a partir de un estado inicial σ que satisface p, alcanza un estado σ' en una posición denotada por una aserción r, entonces σ' satisface r (este enunciado se conoce como Lema de Preservación Composicional). En este caso se habla de sensatez *fuerte*. No vamos a desarrollar la prueba completa de la sensatez fuerte de H; como ejemplo demostramos a continuación el caso de la selección condicional (el resto de la prueba queda como ejercicio). Sea la siguiente *proof outline* estándar de {p} S {q}:

$$\{p\}\ldots\{pre(T)\}\ \text{if } B \text{ then } \{pre(S_1)\}\ S_1\ \{post(S_1)\}$$
$$\text{else } \{pre(S_2)\}\ S_2\ \{post(S_2)\}\ \text{fi}\ \{post(T)\}\ldots\{q\}$$

Supongamos que a partir de un estado inicial σ_0, la computación $\pi(S, \sigma_0)$ tiene la forma

$$(S, \sigma_0) \rightarrow^* (U, \sigma_1) \rightarrow (V, \sigma_2) \rightarrow^* \ldots$$

Sea T = if B then S_1 else S_2 fi, la primera instrucción de la continuación sintáctica U, y supongamos que $\sigma_1(B)$ = verdadero. Por lo tanto, S_1 es la primera instrucción de la continuación sintáctica V, y $\sigma_2 = \sigma_1$. Por la definición de *proof outline* se cumple (pre(T) $\wedge$ B) $\rightarrow$ pre(S_1), y por la hipótesis inductiva, $\sigma_1 \models$ pre(T). Veamos que $\sigma_2 \models$ pre(S_1).

Como $\sigma_1(B)$ = verdadero y $\sigma_2 = \sigma_1$, entonces $\sigma_2 \models pre(T) \wedge B$, y por lo tanto se cumple $\sigma_2 \models pre(S_1)$. De la misma manera se puede probar el caso en que la selección condicional sigue por el else.

La demostración de la sensatez del método H* es naturalmente la misma que la de H, salvo el caso de la regla de la repetición.

Teorema 14.2. Sensatez del método H*

Para todo programa S de PLW y todo par de aserciones p y q de Assn, se cumple

$$Tr \vdash_{H^*} \langle p \rangle\ S\ \langle q \rangle \rightarrow \models \langle p \rangle\ S\ \langle q \rangle$$

Consideramos genéricamente una postcondición q, que en particular puede ser true. Para simplificar la notación usaremos de ahora en más $\vdash$ en lugar de $Tr \vdash_{H^*}$ cuando el significado quede claro por contexto.

La implicación ya se probó en el Teorema 14.1 para los axiomas SKIP* y ASI* y las reglas SEC*, COND* y CONS*. En el caso de la regla REP*, hay que demostrar que se cumple $\models \langle \exists n: p(n) \rangle$ while B do S od $\langle p(0) \rangle$ si se prueba en H* la fórmula $\langle p(n + 1) \rangle$ S $\langle p(n) \rangle$ y las aserciones $p(n + 1) \rightarrow B$ y $p(0) \rightarrow \neg B$ están en Tr. Supongamos entonces que vale $\vdash \langle p(n + 1) \rangle$ S $\langle p(n) \rangle$, $p(n + 1) \rightarrow B$, y $p(0) \rightarrow \neg B$. Por hipótesis inductiva, se cumple $\models \langle p(n + 1) \rangle$ S $\langle p(n) \rangle$. Sea σ un estado inicial tal que $\sigma \models \exists n: p(n)$, y supongamos que $val(\pi(\text{while B do S od}, \sigma)) = \perp$, es decir que el while no termina:

- No puede ser que $\sigma \models p(0)$, porque por $p(0) \rightarrow \neg B$ y la semántica de PLW el while terminaría.
- Así, $\sigma \models p(n)$ para algún $n > 0$. Por $p(n + 1) \rightarrow B$, $\models \langle p(n + 1) \rangle$ S $\langle p(n) \rangle$ y la semántica de PLW, entonces la única posibilidad para no alcanzar nunca un estado σ' tal que $\sigma' \models p(0)$ es que exista una cadena descendente infinita en el orden parcial bien fundado (N, <), lo que es absurdo.

De esta manera, el while termina en un estado final σ' que cumple $\sigma' \models p(0)$.

Fin de Teorema

De manera similar se demuestra la sensatez de la regla alternativa REP** (queda como ejercicio).

Como se puede observar en la prueba del Teorema 14.1, la sensatez de H se cumple independientemente de la interpretación considerada. En efecto, H tiene sensatez *total*, lo que se formula de la siguiente manera:

$$Tr_I \vdash_H \{p\}\ S\ \{q\} \rightarrow \models_I \{p\}\ S\ \{q\}, \text{ para toda interpretación I}$$

Tr_I contiene todas las aserciones verdaderas con respecto a la interpretación I. Que valga $\models_I \{p\}\ S\ \{q\}$ significa que $\{p\}\ S\ \{q\}$ es verdadera considerando I. Como en este caso $\{p\}\ S\ \{q\}$ es verdadera para todas las interpretaciones, se dice que la fórmula es *válida*.

En cambio, H* no es totalmente sensato. Por ejemplo, en H* se prueba claramente

$$\langle true \rangle\ S :: \text{while } x > 0 \text{ do } x := x - 1 \text{ od } \langle true \rangle$$

pero semánticamente la fórmula no necesariamente se cumple si se considera un modelo no estándar de los números naturales, en el que a la sucesión infinita inicial asimilable a los números naturales le sigue un conjunto de cadenas de números no estándar, cada una de ellas sin mínimo ni máximo: si el valor inicial de x es un número no estándar, el programa no termina. Para contemplar cualquier interpretación, lo que se suele hacer es extender el lenguaje de especificación con el lenguaje de primer orden de la aritmética de Peano y un predicado unario nat que caracterice a los números naturales, y correspondientemente extender el dominio semántico con los números naturales como modelo del lenguaje de Peano, con las operaciones y relaciones aritméticas habituales (interpretación estándar). Las interpretaciones de este tipo se denominan *aritméticas*. Por el Teorema 14.2, entonces, H* es sensato considerando cualquier interpretación aritmética. Se dice en este caso que el método tiene sensatez *aritmética*. Notar que el problema con el programa S anterior ahora se puede resolver probando en H* la fórmula

$$\langle nat(x) \rangle\ S :: \text{while } x > 0 \text{ do } x := x - 1 \text{ od } \langle true \rangle$$

porque los números no estándar que provocan la no terminación no satisfacen la precondición nat(x).

Pasamos ahora a la demostración de la completitud del método H. Para facilitar el desarrollo de la prueba, primero vamos a asumir que:

- H incluye como axiomas a todas las aserciones verdaderas sobre los números enteros considerando la interpretación estándar.
- Con el lenguaje de especificación Assn se pueden expresar todas las aserciones intermedias de una prueba (se dice en este sentido que Assn es *expresivo* con respecto a PLW y la interpretación estándar de los números enteros). Más precisamente, asumiremos que con Assn se puede expresar la postcondición más fuerte de todo programa S con respecto a toda precondición p. Semánticamente, dicha postcondición denota el conjunto, conocido como post(p, S), de todos los estados finales obtenidos de ejecutar S a partir de todos los estados iniciales que satisfacen p, es decir

$$post(p, S) = \{\sigma' \mid \exists \sigma: \sigma \models p \wedge M(S)(\sigma) = \sigma' \neq \perp\}$$

Se puede definir alternativamente la expresividad de un lenguaje de especificación en términos de la precondición (liberal) más débil, que denota el conjunto pre(S, q) de todos los estados iniciales a partir de los cuales se obtienen, por la ejecución de S, si termina, todos los estados finales que satisfacen q, es decir

$$pre(S, q) = \{\sigma \mid \forall \sigma': M(S)(\sigma) = \sigma' \neq \perp \longrightarrow \sigma' \models q\}$$

Luego analizaremos qué sucede con la completitud de H omitiendo las dos asunciones anteriores.

Teorema 14.3. Completitud del método H

Para todo programa S de PLW y todo par de aserciones p y q de Assn, se cumple

$$\models \{p\}\ S\ \{q\} \rightarrow Tr \vdash_H \{p\}\ S\ \{q\}$$

Lo probaremos por inducción sobre la estructura de S, considerando sus cinco formas posibles (skip, asignación, secuencia, selección condicional y repetición). La idea básica

es, asumiendo que la conclusión de una regla es verdadera, mostrar que las premisas también lo son, y así por hipótesis inductiva, que se pueden probar en H.

1. Si $\models \{p\}$ skip $\{q\}$, entonces $\vdash \{p\}$ skip $\{q\}$.
   Por la semántica de PLW, (skip, σ) $\rightarrow$ (E, σ). Como $\models \{p\}$ skip $\{q\}$, si $\sigma \models p$ entonces $\sigma \models q$, y por lo tanto $p \rightarrow q$. La siguiente es una prueba de $\{p\}$ skip $\{q\}$:
   1. $\{q\}$ skip $\{q\}$ por SKIP, 2. $p \rightarrow q$ por MAT, 3. $\{p\}$ skip $\{q\}$ por CONS, 1, 2.
2. Si $\models \{p\}$ x := e $\{q\}$, entonces $\vdash \{p\}$ x := e $\{q\}$.
   Por la semántica de PLW, (x := e, σ) $\rightarrow$ (E, σ[x | e]). Como $\models \{p\}$ x := e $\{q\}$, si $\sigma \models p$ entonces $\sigma[x \mid e] \models q$, y también $\sigma \models q[x \mid e]$ por el Lema de Sustitución, por lo que $p \rightarrow q[x \mid e]$. La siguiente es una prueba de $\{p\}$ x := e $\{q\}$:
   1. $\{q[x \mid e]\}$ x := e $\{q\}$ por ASI, 2. $p \rightarrow q[x \mid e]$ por MAT, 3. $\{p\}$ x := e $\{q\}$ por CONS, 1, 2.
3. Si $\models \{p\}\ S_1 ; S_2\ \{q\}$, entonces $\vdash \{p\}\ S_1 ; S_2\ \{q\}$.
   Sea r una aserción intermedia entre S_1 y S_2 que denota el conjunto post(p, S_1). La aserción r se puede expresar por la asunción de expresividad del lenguaje Assn. Como $\models \{p\}\ S_1 ; S_2\ \{q\}$, por la definición de post(p, S_1) y la semántica de PLW se cumple $\models \{p\}\ S_1\ \{r\}$ y $\models \{r\}\ S_2\ \{q\}$. Por hipótesis inductiva, $\vdash \{p\}\ S_1\ \{r\}$ y $\vdash \{r\}\ S_2\ \{q\}$. Finalmente, aplicando la regla SEC se llega a $\vdash \{p\}\ S_1 ; S_2\ \{q\}$.
4. Si $\models \{p\}$ if B then S_1 else S_2 fi $\{q\}$, entonces $\vdash \{p\}$ if B then S_1 else S_2 fi $\{q\}$. La prueba es similar a la del item anterior y queda como ejercicio.
5. Si $\models \{r\}$ while B do S od $\{q\}$, entonces $\vdash \{r\}$ while B do S od $\{q\}$.
   Suponiendo $\models \{r\}$ while B do S od $\{q\}$, hay que encontrar un invariante p que cumpla $r \rightarrow p$, $(p \wedge \neg B) \rightarrow q$, y $\models \{p \wedge B\}\ S\ \{p\}$. De esta manera, por hipótesis inductiva y aplicando las reglas REP y CONS, se llega a $\vdash \{r\}$ while B do S od $\{q\}$. Semánticamente, p debe denotar el conjunto de todos los estados alcanzados, a partir de uno inicial σ_0 que satisfaga r, por cualquier cantidad de iteraciones de S, es decir el conjunto C siguiente:

$$C = \{\sigma \mid \exists k, \sigma_0, \ldots, \sigma_k: \sigma = \sigma_k \wedge \sigma_0 \models r \wedge (\forall i < k: M(S)(\sigma_i) = \sigma_{i+1} \wedge \sigma_i(B) = \text{verdadero})\}$$

Una forma razonable de p sería

$$p = p_0 \vee \ldots \vee p_k \vee \ldots$$

tal que $p_0 = r$, y p_{i+1} denota post($p_i \wedge B$, S) para todo $i \geq 0$. Las aserciones p_1, p_2, …, son expresables por la asunción de expresividad. Veamos que p también es expresable. Sea $\{y_1, \ldots, y_n\}$ el conjunto formado por las variables de S más las variables libres de r y q. Sea $\{z_1, \ldots, z_n\}$ un conjunto de igual tamaño con nuevas variables. Sea S* :: while $B \wedge (y_1 \neq z_1$ v … v $y_n \neq z_n)$ do S od. Y sea p* una aserción que denota post(r, S*), expresable por la asunción de expresividad. Notar que la transformación de estados asociada a S* es la misma que la del while original, y que eligiendo adecuadamente las z_i se puede forzar la terminación de S* sin que valga $\neg B$. Se cumple que el invariante buscado es

$$p = \exists z_1 \ldots z_n : p^*$$

Efectivamente, p satisface las condiciones establecidas previamente:

- $r \rightarrow p$, eligiendo $z_i = y_i$ para todo i (S* termina después de cero iteraciones).
- $(p \wedge \neg B) \rightarrow q$, porque si vale $p^* \wedge \neg B$, las y_i tienen los valores finales del while original, y como |= {r} while B do S od {q}, entonces vale q.
- |= {$p \wedge B$} S {p}. Si σ |= $p^* \wedge B$, los $\sigma(y_i)$ se obtuvieron al ejecutar S una cantidad finita de pasos, quedando $\sigma(B)$ = verdadero, por lo que eligiendo $z_i = M(S)(\sigma)(y_i)$ para todo i, vale $M(S)(\sigma)$ |= p.

Fin de Teorema

De esta manera, queda formalizado que axiomas como el de invariancia y reglas como las de la disyunción y la conjunción, son redundantes en el método H.
La demostración de la completitud del método H* es la misma que la de H, sin considerar la regla de la repetición.

Teorema 14.4. Completitud del método H*

Para todo programa S de PLW y todo par de aserciones p y q de Assn, se cumple

$$\models \langle p \rangle\ S\ \langle q \rangle \rightarrow Tr \vdash_{H^*} \langle p \rangle\ S\ \langle q \rangle$$

Como en el Teorema 14.2, consideramos genéricamente una postcondición q, que en particular puede ser true. La implicación ya se probó en el Teorema 14.3 para el skip, la asignación, la secuencia y la selección condicional, utilizando inducción estructural. En el caso de la repetición, asumiendo $\models \langle r \rangle$ while B do S od $\langle q \rangle$ hay que encontrar un invariante parametrizado p(n) que cumpla las premisas de la regla REP*. Semánticamente, p(n) debe denotar el siguiente conjunto C de estados:

$$C = \{\sigma \mid \sigma \models nat(n) \wedge \sigma(n) = k \wedge \exists \sigma_0, \ldots, \sigma_k: \sigma = \sigma_0 \wedge \sigma_k \models q \wedge \neg B \wedge (\forall i < k: M(S)(\sigma_i) = \sigma_{i+1} \wedge \sigma_i(B) = \text{verdadero})\}$$

En palabras, los estados de C son aquéllos a partir de los cuales el programa while B do S od termina en exactamente k iteraciones y los estados finales satisfacen q. Claramente, p(n) satisface las premisas de REP*. Por ejemplo, para ver que $\models \langle p(n+1) \rangle$ S $\langle p(n) \rangle$, notar que p(n + 1) implica la existencia de una secuencia de estados $\sigma_0, \ldots, \sigma_{n+1}$, y que luego de la ejecución de S se cumple p(n) considerando la secuencia $\sigma_1, \ldots, \sigma_{n+1}$. No vamos a definir la sintaxis de p(n), la idea es similar a la planteada en el Teorema 14.3. Finalmente, por la asunción, la hipótesis inductiva y la aplicación de las reglas REP* y CONS*, se obtiene $\vdash \langle r \rangle$ while b do S od $\langle q \rangle$.

Fin de Teorema

Considerando la regla alternativa REP**, la prueba de la completitud de H* debe incluir la demostración de la expresividad de Assn con respecto a la función cota. Dado un programa S :: while B do S od, y un estado inicial σ, sea iter(S, σ) una función parcial que define el número de iteraciones de S a partir de σ. Claramente iter es computable, el siguiente programa calcula la función:

S_x :: x := 0 ; while B do x := x + 1 ; S od

Para garantizar la completitud de H* en este caso, se requiere que todas las funciones computables sean expresables.

Analizando la completitud en un marco más general, cabe remarcar que el método H por sí solo es incompleto. Por ejemplo, para toda interpretación I se cumple $Tr_I \models \{true\}\ x := e\ \{x = e\}$, cuando $x \notin var(e)$, pero contando solamente con los axiomas y reglas de H no puede probarse $true \rightarrow e = e$. No es solución ampliar H con un sistema deductivo asociado a la interpretación considerada: es razonable que H trate mínimamente con los números enteros, y por el Teorema de Incompletitud de Gödel sabemos que las aserciones verdaderas sobre los mismos no conforman un conjunto recursivamente numerable, por lo que tampoco lo es el conjunto de fórmulas {{true} skip {p}}. Ni restringiendo las aserciones a true y false se soluciona el problema: por la indecidibilidad del problema de la detención (en el marco de los programas PLW y los números enteros) el conjunto de fórmulas {{true} S {false}} tampoco es recursivamente numerable.

Así llegamos al concepto de completitud *relativa* de H (y de H*): se le agregan a H todas las aserciones verdaderas con respecto a la interpretación considerada. No es habitual utilizar sistemas deductivos con un conjunto de axiomas de esta naturaleza, pero hay que tener en cuenta que de lo que se trata es de probar programas, no enunciados del dominio en que se ejecutan. De esta manera, como vimos en los ejemplos, en las pruebas se asume la existencia de un oráculo que provee las aserciones verdaderas necesarias (manipuladas mediante la regla de consecuencia).

El requerimiento de expresividad del lenguaje de especificación con respecto al lenguaje de programación y la interpretación, refuerza el concepto de completitud relativa (en este caso se habla también de completitud *en el sentido de Cook*). Una típica estructura que asegura la expresividad en lenguajes de primer orden como Assn con programas del tipo PLW es el modelo estándar de la aritmética de Peano, al que ya nos hemos referido. Con dicha estructura se pueden codificar computaciones de programas mediante números naturales. Así, se pueden expresar las postcondiciones más fuertes: simples elementos del dominio representan conjuntos de estados intermedios con determinadas propiedades. Como contraejemplo, la aritmética de Presburger, que no tiene la multiplicación, no sirve para la expresividad requerida.

Por lo tanto, tomando como base el Teorema 14.3 se puede formular más en general que

$$\models_I \{p\}\ S\ \{q\} \rightarrow Tr_I \vdash_H \{p\}\ S\ \{q\}$$

para toda interpretación I, tal que el lenguaje de especificación es expresivo con respecto a PLW e I. Por su parte, tomando como base el Teorema 14.4 se puede formular más en general que

$$\models_{I^+} \langle p \rangle\ S\ \langle q \rangle \rightarrow Tr_{I^+} \vdash_{H^*} \langle p \rangle\ S\ \langle q \rangle$$

para toda interpretación aritmética I^+. Se define en este sentido que la completitud de H* es *aritmética*.

Ejercicios de la Clase 14

1. Completar la prueba del Teorema 14.1.
2. Probar que el axioma y la regla de invariancia (INV), la regla de la disyunción (OR) y la regla de la conjunción (AND), preservan la sensatez de H.
3. Mostrar que la regla alternativa para REP* y la regla REP** son sensatas.
4. Determinar cuáles de las siguientes reglas son sensatas:
   i. $$\frac{\{p_1\}\ S\ \{q_1\}\ ,\ \{p_2\}\ S\ \{q_2\}}{\{p_1 \vee p_2\}\ S\ \{q_1 \vee q_2\}}$$
   ii. Regla del SHIFT:
   $$\frac{\{p \wedge q\}\ S\ \{r\},\ free(q) \cap change(S) = \varnothing}{\{p\}\ S\ \{q \rightarrow r\}}$$
   iii. Volviendo a la instrucción repeat S until B considerada en el Ejercicio 4 de la Clase 11:
   $$\frac{\{p \wedge \neg B\}\ S\ \{p\}}{\{p\}\ \text{repeat S until B}\ \{p \wedge B\}}$$
   iv. $$\frac{\langle p \wedge B \rangle\ S\ \langle p \rangle\ ,\ p \rightarrow B}{\langle p \rangle\ \text{while B do S od}\ \langle true \rangle}$$
5. Completar la prueba de la sensatez fuerte del método H (en la Clase 14 sólo se desarrolló el caso de la selección condicional).
6. Probar que el sistema que tiene como axiomas a todas las aserciones verdaderas sobre los números naturales, considerando la suma y la multiplicación, es completo. ¿Por qué esto no contradice el Teorema de Incompletitud de Gödel?

7. Completar la prueba del Teorema 14.3.
8. Probar que el axioma de asignación “hacia adelante” que se presentó en la Clase 12: $\{p\}\ x := e\ \{\exists z: p[x \mid z] \wedge x = e[x \mid z]\}$, es redundante en H.
9. Probar que también el axioma y la regla INV, la regla OR y la regla AND, son redundantes en H.
10. Probar que $\models \{p\}\ S\ \{q\} \leftrightarrow \{\sigma \mid \sigma \models p\} \subseteq pre(S, q) \leftrightarrow post(p, S) \subseteq \{\sigma \mid \sigma \models q\}$.

Clase 15. Misceláneas de verificación de programas

TEMA 15.1. VERIFICACIÓN DE PROGRAMAS CON ARREGLOS

Extendiendo el lenguaje PLW con variables de tipo *arreglo*, tenemos que adecuar el mecanismo de sustitución para que el axioma de asignación de la metodología de verificación de programas descripta siga siendo verdadero, como mostramos en lo que sigue.

Consideramos sólo arreglos unidimensionales, por ejemplo a[1:N], con elementos a[1], …, a[N]. Los índices pueden ser expresiones enteras.

Veamos, antes de seguir con las definiciones, un ejemplo que muestra cómo el axioma de asignación deja de ser verdadero si permitimos referencias anidadas, es decir si los índices de los arreglos pueden incluir variables suscriptas. Aplicando ASI y CONS se obtiene

$$\{true\}\ a[i] := 1\ \{a[i] = 1\}$$

Pero esta fórmula no es verdadera. Por ejemplo, semánticamente se cumple

$$\{a[1] = 2 \wedge a[2] = 2\}\ a[a[2]] := 1\ \{a[a[2]] = 2\}$$

que contradice la fórmula anterior. Lo que sucede es que el elemento a[a[2]] de la asignación no es el mismo que el de la postcondición. Para simpificar, no vamos a permitir referencias anidadas.

Ahora un estado asigna números enteros también a variables suscriptas, es decir a pares (a, i), donde a es una variable de tipo arreglo, e i es una constante entera. Se define

$$\sigma(a[e]) = \sigma((a, \sigma(e)))$$

Por lo tanto, expresiones como $\sigma(a[x + y]) > 0$ deben interpretarse como $\sigma((a, \sigma(x) + \sigma(y))) > 0$. El variante de un estado, considerando variables de tipo arreglo, se define de la siguiente manera. Dadas dos variables a y b:

- $\sigma[a[e] \mid e'](a, i)) = \sigma(e')$, si $\sigma(e) = i$
- $\sigma[a[e] \mid e'](a, i)) = \sigma((a, i))$, si $\sigma(e) \neq i$
- $\sigma[a[e] \mid e'](b, i)) = \sigma((b, i))$

La semántica de la asignación a variables de tipo arreglo se especifica mediante la relación $\rightarrow$ de este modo:

$$(a[e] := e', \sigma) \rightarrow (E, \sigma[a[e] \mid e'])$$

Lamentablemente, aún eliminando las referencias anidadas seguimos teniendo problemas con el axioma de asignación. Por ejemplo, aplicando ASI y CONS se obtiene

$$\{a[y] = 1\}\ a[x] := 0\ \{a[x] + 1 = a[y]\}$$

Pero esta fórmula no es verdadera. Por ejemplo, semánticamente se cumple

$$\{a[1] = 1 \wedge x = 1 \wedge y = 1\}\ a[x] := 0\ \{a[x] + 1 \neq a[y]\}$$

que contradice la fórmula anterior. En este caso, el problema es que la asignación a a[x] también modifica a[y], porque son el mismo elemento (los índices x e y son iguales). Esto se conoce como problema de *alias*.

Más en general, el problema está en cómo se aplica la sustitución, que es el mecanismo que captura el efecto de una asignación. En lo que sigue presentamos una adecuación habitual de la sustitución para restablecer la aplicabilidad del axioma de asignación.

La idea básica es utilizar una expresión condicional para definir las sustituciones, de modo tal que éstas se resuelvan ya no en tiempo de sustitución sino en tiempo de evaluación (considerando un determinado estado). La expresión condicional tiene la forma cond(B, e_1, e_2), donde B es una igualdad entre expresiones, y e_1 y e_2 son expresiones. Se define así:

$$\sigma(\text{cond}(e = e', e_1, e_2)) = e_1, \text{ si } \sigma(e) = \sigma(e')$$
$$= e_2, \text{ si } \sigma(e) \neq \sigma(e')$$

Notar que la evaluación devuelve una expresión, no un valor. Justamente, la idea es resolver la sustitución semánticamente en el momento oportuno, no de manera puramente sintáctica. Volviendo al último ejemplo, con este mecanismo, a pesar de la diferencia sintáctica entre a[x] y a[y], si se cumple $\sigma(x) = \sigma(y)$ se aplica efectivamente la sustitución correspondiente. Enseguida seguimos con el ejemplo.
Utilizando la expresión condicional, se define la sustitución de variables de tipo arreglo de la siguiente manera (naturalmente, el único caso que amerita consideración especial es la sustitución de una variable suscripta por otra):

$$a[e]\ [a[e_1] \mid e_2] = cond(e = e_1, e_2, a[e])$$

Así, el test de igualdad entre los índices queda registrado en la expresión condicional resultante (si se permitieran referencias anidadas se debería utilizar una alternativa más compleja). Completando la adecuación del axioma de asignación, queda por extender el alcance del Lema de Sustitución a las variables de tipo arreglo:

$$\sigma[a[e] \mid e'] \models p \leftrightarrow \sigma \models p[a[e] \mid e']$$

De esta manera se restituye la sensatez de los métodos H y H*. Veamos cómo la adecuación de la sustitución resuelve el problema de alias anterior. Dada la fórmula

$$\{p\}\ a[x] := 0\ \{a[x] + 1 = a[y]\}$$

aplicando ASI con la sustitución de variables suscriptas, la precondición p, es decir $(a[x] + 1 = a[y])\ [a[x] \mid 0]$, queda de la siguiente forma:

$$(cond(x = x, 0, a[x]) + 1 = cond(y = x, 0, a[y]))$$

Notar que para todo estado inicial σ, si $\sigma \models p$, entonces $\sigma \models x \neq y$. O dicho de otra manera, la asignación a[x] := 0 no puede tener postcondición a[x] + 1 = a[y] si la precondición implica x = y.

TEMA 15.2. DESARROLLO SISTEMÁTICO DE PROGRAMAS

En esta sección mostramos un ejemplo muy sencillo de desarrollo sistemático de programas, guiado por la metodología de verificación presentada. La idea es desarrollar un programa en simultáneo con la construcción de su prueba de correctitud. El foco se pone en la construcción de un while, a partir de un invariante p y una función cota t (nos basaremos en la regla alternativa REP** de H*). Supongamos que se quiere desarrollar un programa P de PLW con la siguiente estructura:

P :: T ; while B do S od

tal que cumpla $\langle r \rangle$ P $\langle q \rangle$. De acuerdo a la metodología de prueba, P debe satisfacer los siguientes cinco requerimientos:

1. Las inicializaciones de T deben establecer el invariante p: $\{r\}$ T $\{p\}$.
2. La aserción p debe ser efectivamente un invariante del while: $\{p \wedge B\}$ S $\{p\}$.
3. Al finalizar el while debe valer la postcondición q: $(p \wedge \neg B) \rightarrow q$.
4. La función cota debe decrecer con cada iteración: $\{p \wedge B \wedge t = Z\}$ S $\{t < Z\}$, siendo Z una variable de especificación.
5. Mientras se cumpla el invariante, la función cota no debe ser negativa: $p \rightarrow t \geq 0$.

La siguiente *proof outline* genérica refleja los requerimientos planteados (recordar que no se suelen incluir los requerimientos 4 y 5, sino que se documenta directamente la función cota):

$\langle r \rangle$ T ; $\langle$inv: p , fc: t$\rangle$ while B do $\langle p \wedge B \rangle$ S $\langle p \rangle$ od $\langle p \wedge \neg B \rangle$ $\langle q \rangle$

Para evitar el desarrollo de un programa trivial que no sea el pretendido, se debe establecer que determinadas variables no sean modificadas. Por ejemplo, la especificación del programa del factorial del Ejemplo 12.3 es (x > 0, y = x!). Si se permitiera modificar x, una solución trivial sería S_{fac} :: x := 1 ; y := 1.

Ejemplo 15.1. Desarrollo sistemático de un programa PLW

Vamos a construir un programa S_{sum} que calcula en una variable x la suma de los elementos de un arreglo de valores enteros a[0:N – 1], con $N \geq 0$. Se establece que $a \notin change(S_{sum})$, por lo que evitamos la posibilidad de que se construya un programa inadecuado y al mismo tiempo las complicaciones técnicas producto del uso de arreglos, como se mostró en la sección anterior. Por convención, si N = 0 la suma es cero. S_{sum} tendrá la forma

S_{sum} :: T ; while B do S od

y satisfará la especificación (r, q), siendo

$r = N \geq 0$

$q = (x = \Sigma_{i=0,N-1}\, a[i])$

El primer paso es encontrar un invariante p para el while. Una estrategia conocida, ya mencionada, es generalizar la postcondición q, reemplazando constantes por variables. En este caso reemplazamos N por una variable k, y proponemos

$p = (0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i])$

En lo que sigue definimos una expresión booleana B, un cuerpo S y una función cota t apropiados para satisfacer los cinco requerimientos planteados previamente:

Para que se cumpla 1: {r} T {p}, elegimos T :: k := 0 ; x := 0.
Para que se cumpla 3: $(p \wedge \neg B) \rightarrow q$, elegimos $B = (k \neq N)$.
Para que la función cota t decrezca con cada iteración y se mantenga mayor o igual que cero (requerimientos 4 y 5), hacemos que k := k + 1 sea parte del cuerpo S del while y elegimos t = N – k.
De esta manera, la *proof outline* tiene por ahora la siguiente forma:

$\langle N \geq 0 \rangle$
  k := 0 ; x := 0 ;
$\langle$inv: $0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i]$, fc: $N - k\rangle$

while $k \neq N$ do

$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k \neq N \rangle$

S'

$\langle (0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i])\ [k \mid k + 1] \rangle$

k := k + 1

$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \rangle$

od

$\langle 0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k = N \rangle$

$\langle x = \Sigma_{i=0,N-1}\, a[i] \rangle$

Notar cómo se descompuso el cuerpo S del while en la secuencia S' ; k := k + 1. La aserción intermedia entre S' y k := k + 1 se obtuvo aplicando el axioma ASI. El subprograma S', entonces, debe satisfacer

$$\{0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] \wedge k \neq N\}\ S'\ \{0 \leq k + 1 \leq N \wedge x = \Sigma_{i=0,k}\, a[i]\}$$

Se cumple que la precondición de S' implica la aserción a_1 siguiente:

$$0 \leq k + 1 \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i]$$

También se cumple que la postcondición de S' implica la aserción a_2 siguiente:

$$0 \leq k + 1 \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i] + a[k]$$

Como $\{a_1\}$ x : = x + a[k] $\{a_2\}$, entonces elegimos S' :: x := x + a[k] para que se cumpla el requerimiento 2 que faltaba, es decir $\{p \wedge B\}$ S $\{p\}$, y de esta manera concluimos la construcción de S_{sum}, en simultáneo con su prueba de correctitud total. La *proof outline* quedó de la siguiente manera:

$\langle N \geq 0 \rangle$

k := 0 ; x := 0 ;

$\langle$inv: $0 \leq k \leq N \wedge x = \Sigma_{i=0,k-1}\, a[i]$, fc: $N - k\rangle$

while $k \neq N$ do x := x + a[k] ; k := k + 1 od

$\langle x = \Sigma_{i=0,N-1}\, a[i] \rangle$

Fin de Ejemplo

TEMA 15.3. VERIFICACIÓN DE PROGRAMAS CON PROCEDIMIENTOS

En lo que sigue describimos aspectos relevantes de la verificación de programas con procedimientos. Introducimos los conceptos gradualmente, comenzando simplemente con el mecanismo de invocación, luego pasando por la recursión, y finalmente terminando con el uso de parámetros. Extendemos entonces inicialmente el lenguaje PLW con procedimientos no recursivos y sin parámetros. La nueva sintaxis es

P :: procedure <nombre$_1$>: S_1, …, procedure <nombre$_n$>: S_n ; S

Los S_i son los cuerpos de los procedimientos, y S es el cuerpo principal del programa P. Las instrucciones son las conocidas más la instrucción de *invocación* call proc, siendo proc el nombre de un procedimiento. La ejecución de call proc consiste en la ejecución del cuerpo S_i de proc. Si S_i termina, el programa continúa con la instrucción siguiente al call. Un procedimiento puede invocar a otro, con la restricción (por ahora) de que el grafo de invocaciones sea acíclico, de manera de evitar la recursión tanto simple (que un procedimiento se invoque a sí mismo) como mutua (que al menos intervengan dos procedimientos). El siguiente es un primer ejemplo de programa con procedimientos. Utiliza un procedimiento suma1 para sumar dos números:

P_1 :: procedure suma1: x := x + 1 ;
    while y > 0 do call suma1 ; y := y – 1 od

La semántica formal de la invocación se define mediante la relación $\rightarrow$ de este modo:

$(\text{call proc}, \sigma) \rightarrow (S, \sigma)$

tal que S es el cuerpo de proc. En realidad, las configuraciones tienen ahora tres componentes, porque a la continuación sintáctica y el estado corriente se le agrega el *entorno* (*environment*), que contiene las declaraciones de los procedimientos. Como las transiciones no modifican el entorno, se lo puede omitir para simplificar la notación.

La regla de prueba habitual asociada a la instrucción de invocación que extiende H se denomina INVOC:

$$\text{Regla INVOC} \qquad \frac{\{p\}\ S\ \{q\}}{\{p\}\ \text{call proc}\ \{q\}}$$

tal que S es el cuerpo de proc. La sensatez y completitud del método H extendido con INVOC se preservan, y se prueban fácilmente a partir de la sensatez y completitud de H y la semántica del call (la prueba queda como ejercicio).

Ejemplo 15.2. Correctitud parcial con un procedimiento no recursivo

Vamos a verificar la correctitud parcial del programa P_1 anterior. Se quiere probar

$\{x = X \wedge y = Y \geq 0\}$
P_1 :: procedure suma1: x := x + 1 ;
while y > 0 do call suma1 ; y := y – 1 od
$\{x = X + Y\}$

Primero consideramos el procedimiento suma1, utilizando una variable de especificación Z:

1. $\{x = Z \wedge y \geq 1\}$ x := x + 1 $\{x = Z + 1 \wedge y \geq 1\}$ (ASI, CONS)

La condición $y \geq 1$ es necesaria para vincular esta fórmula con el resto de la prueba (se establece que el procedimiento se ejecuta con $y \geq 1$). Para probar el cuerpo principal de P_1 empleamos la regla de instanciación INST en combinación con INVOC, obteniendo una especificación de la invocación expresada en términos de las variables de la especificación del programa:

2. $\{x = X + Y – y \wedge y \geq 1\}$ call suma1 $\{x = X + Y – y + 1 \wedge y \geq 1\}$ (1, INVOC, INST)

La regla INST es muy útil en las pruebas de programas con procedimientos. Notar que la instanciación de Z con X + Y – y es posible porque $y \notin$ var(suma1). Utilizando como

invariante del while la aserción p = (x = X + Y – y ∧ y ≥ 0), se llega sin problemas a {x = X ∧ y = Y ≥ 0} P_1 {x = X + Y}. Queda como ejercicio completar la prueba.

Fin de Ejemplo

Extendemos ahora PLW con procedimientos recursivos. Para simplificar, consideramos sólo la recursión simple. En este caso la regla INVOC obviamente no sirve, porque para probar {p} call proc {q} requiere probar {p} S {q}, y si S invoca a proc, se produce una circularidad que la regla no puede resolver. Una solución habitual para tratar este tipo de recursión consiste en desarrollar una prueba por *asunción y descarte*: se asume que se cumple la especificación de una invocación interna, se prueba a partir de dicha asunción que la misma especificación se cumple con respecto al cuerpo del procedimiento que incluye la invocación, y luego se descarta la asunción. De esta manera se define una regla con una forma particular (es una meta-regla), con una premisa que no es una fórmula de correctitud sino un enunciado acerca de la existencia de una prueba con una asunción. La regla se denomina REC:

Regla REC

$$\frac{\{p\}\ \text{call proc}\ \{q\} \vdash \{p\}\ S\ \{q\}}{\{p\}\ \text{call proc}\ \{q\}}$$

tal que S es el cuerpo de proc. El call proc de la premisa debe entenderse como interno de S, y el call proc de la conclusión como invocante de S. Una vez aplicada REC se descarta la asunción {p} call proc {q}. El método H extendido con REC sigue siendo sensato. Dado que la premisa de REC no es una fórmula de correctitud, la sensatez de la regla no puede probarse como antes. En este caso la inducción se plantea de un modo algo distinto: asumiendo que se obtiene una fórmula verdadera mediante una prueba determinada, se demuestra que la misma fórmula es verdadera mediante una prueba más larga, medida ahora en invocaciones recursivas. La completitud del método también se conserva, y se prueba en base a la posibilidad de expresar una aserción que denota el conjunto de todos los estados alcanzados después de una cantidad arbitraria de invocaciones recursivas.

Ejemplo 15.3. Correctitud parcial con procedimientos recursivos

El siguiente programa calcula el factorial mediante un procedimiento recursivo fact:

P_2 :: procedure fact: if x = 0 then y := 1
else x := x – 1 ; call fact ; x := x + 1 ; y := y . x fi ;
call fact

Se quiere probar $\{x \geq 0\}\ P_2\ \{y = x!\}$. Empleando la regla REC, se parte de la asunción $\{x \geq 0\}$ call fact $\{y = x!\}$ y se llega sin problemas a $\{x \geq 0\}\ S\ \{y = x!\}$, siendo S el cuerpo de fact. Queda como ejercicio desarrollar la prueba.
Este otro programa calcula por medio de un procedimiento recursivo pot2 (muy similar a fact), a partir de un número natural x, la potencia x-ésima de 2, es decir 2^x:

P_3 :: procedure pot2: if x = 0 then y := 1
else x := x – 1 ; call pot2 ; x := x + 1 ; y := y . 2 fi ;
call pot2

Supongamos que se quiere probar simplemente $\{x = X\}\ P_3\ \{x = X\}$, es decir que P_3 preserva el valor inicial de x. Empleando la regla REC, hay que probar la premisa

$\{x = X\}$ call pot2 $\{x = X\} \vdash \{x = X\}\ S\ \{x = X\}$

siendo S el cuerpo de pot2. En realidad, dicha premisa debe entenderse como

$\forall X$: $\{x = X\}$ call pot2 $\{x = X\} \vdash \forall X$: $\{x = X\}\ S\ \{x = X\}$

porque no existe ninguna relación entre las X a la izquierda y las X a la derecha del símbolo $\vdash$. Lo que verdaderamente se está pidiendo probar es que asumiendo que la invocación a pot2 preserva el valor de x, se prueba también que S preserva el valor de x, no importa el valor de x en cada caso. Como mostramos a continuación, otra vez debemos recurrir a la regla INST. Los primeros pasos de la prueba son

1. $\{x = Z\}$ call pot2 $\{x = Z\}$ (ASUNCIÓN)
2. $\{x = X\}\ y := y . 2\ \{x = X\}$ (ASI)
3. $\{x = X – 1\}\ x := x + 1\ \{x = X\}$ (ASI, CONS)
4. $\{x = X – 1\}$ call pot2 $\{x = X – 1\}$ (1, INST)

Se llega sin problemas a $\{x = X\}\ P_3\ \{x = X\}$. Queda como ejercicio completar la prueba.

Fin de Ejemplo

La recursión es otra fuente de no terminación de los programas PLW. La prueba habitual de terminación de un procedimiento recursivo también se basa en la definición de un invariante parametrizado p(n). La regla correspondiente, denominada REC*, tiene la siguiente forma:

$$\text{Regla REC*} \qquad \frac{\forall n: (\langle p(n)\rangle \text{ call proc } \langle q\rangle \vdash \langle p(n+1)\rangle\ S\ \langle q\rangle)\ ,\ \neg p(0)}{\langle \exists n: p(n)\rangle \text{ call proc } \langle q\rangle}$$

tal que S es el cuerpo de proc, $n \notin var(S)$ y $n \notin free(q)$. Como REC, REC* es una meta-regla, su primera premisa no es una fórmula de correctitud. La cuantificación en la primera premisa indica que el variante n es el mismo a izquierda y derecha del símbolo $\vdash$. Como antes, el call proc de la premisa debe entenderse como interno de S, y el call proc de la conclusión como invocante de S.

Se define que una computación de una invocación a un procedimiento recursivo proc con cuerpo S es *(q, n)-profunda*, si termina en un estado que satisface q y en todo momento hay a lo sumo n invocaciones activas de proc (una invocación está activa si la instancia correspondiente no ha terminado). En este sentido, la asunción $\langle p(n)\rangle$ call proc $\langle q\rangle$ establece que a partir de cualquier estado que satisface p(n), una computación de call proc es (q, n)-profunda, y se utiliza para probar $\langle p(n+1)\rangle\ S\ \langle q\rangle$, es decir que a partir de cualquier estado que satisface p(n + 1), una computación de call proc es (q, n + 1)-profunda. Como $\neg p(0)$ significa que ningún estado satisface p(0), entonces el cumplimiento de las premisas de REC* asegura la terminación de la invocación call proc a partir de cualquier estado que satisfaga el invariante parametrizado p(n) para algún n.

La sensatez y completitud de H* extendido con REC* se prueban como antes. En el primer caso se asume no terminación y se llega a un absurdo relacionado con el orden parcial bien fundado (N, <), y en el caso de la completitud se demuestra que con una interpretación aritmética se puede expresar p(n).

Con la regla REC* podemos completar la verificación de la correctitud total del programa P_2 anterior que calcula recursivamente el factorial (ver Ejemplo 15.3). Se prueba sin problemas $\langle x \geq 0 \rangle\ P_2\ \langle true \rangle$ (se puede utilizar, por ejemplo, el invariante parametrizado $p(n) = (x \geq 0 \wedge n = x + 1)$). La prueba queda como ejercicio.

Finalmente, pasamos a tratar ahora los procedimientos con parámetros. Restringimos el análisis a dos tipos de pasajes de parámetros, por *valor* y por *resultado*. Los parámetros enviados por valor proveen valores al procedimiento y no son modificables dentro del cuerpo del mismo. Los parámetros devueltos por resultado transfieren valores del procedimiento al entorno de invocación. La declaración de un procedimiento con parámetros en PLW tiene la siguiente forma:

procedure <nombre>(val $y_1, \ldots, y_m$; res $z_1, \ldots, z_n$): S

y la sintaxis de la invocación correspondiente es

call <nombre>($e_1, \ldots, e_m$; $x_1, \ldots, x_n$)

Las y_i y z_i son los parámetros formales por valor y resultado, respectivamente. Las expresiones e_i son los parámetros reales por valor, y las x_i son los parámetros reales por resultado. Las y_i y z_i son dos a dos distintas. Las x_i son también dos a dos distintas y no aparecen en las expresiones e_i. Además, se cumple que $\{y_1, \ldots, y_m\} \cap change(S) = \varnothing$. El conjunto change(S) incluye las variables de S modificables por asignaciones, y los parámetros reales por resultado de las invocaciones desde S. Para simplificar, asumimos que en los procedimientos no hay variables locales ni referencias a variables globales. Tampoco vamos a permitir invocaciones con alias. El uso de alias atenta contra la sensatez del método de verificación, como mostramos a continuación. Sea el procedimiento suma1, ahora con parámetros:

procedure suma1(val y ; res z): z := y + 1

y supongamos que se lo invoca con call suma1(x ; x). Una postcondición natural para suma1 es z = y + 1 (luego establecemos restricciones sobre la forma de la postcondición de un procedimiento), y por lo tanto después de la invocación se obtiene la aserción x = x + 1, que es falsa. Existen reglas sensatas (complejas) para el manejo de alias. En

general, mecanismos como éste, el pasaje de parámetros por nombre, el uso de procedimientos como parámetros, las referencias a variables globales, etc., si bien posibilitan esquemas muy flexibles de programación, atentan contra la sensatez y completitud de los métodos de verificación.

La invocación a un procedimiento con parámetros consiste en el pasaje de los parámetros por valor, la ejecución del cuerpo del procedimiento, y la devolución de los parámetros por resultado. Pero antes que nada, como los nombres de los parámetros del procedimiento pueden entrar en conflicto con las variables de programa del entorno de invocación, se crea una nueva instancia para ellos. Cuando el procedimiento termina y se retorna al entorno de invocación, la instancia se libera. La creación y liberación de instancias se implementan mediante dos operaciones sobre estados, que ahora asignan temporariamente valores a variables. Se conoce como *dominio* o *soporte* de un estado σ, y se denota con dom(σ), al conjunto de variables asignadas por σ. Las operaciones sobre estados son:

- include (x_1, …, x_n ; σ), con $x_i \notin$ dom(σ). La operación agrega las variables x_i al estado σ, el cual les asigna un valor inicial indefinido.
- exclude (x_1, …, x_n ; σ). La operación remueve las x_i del dominio de σ.

La semántica formal de la invocación con parámetros puede especificarse de la siguiente manera (para simplificar la notación, consideramos de ahora en más un solo parámetro por valor y un solo parámetro por resultado):

(call proc(e; x), σ) → (y’ := e; S[y | y’][z | z’]; x := z’; release(y’, z’), include(y’, z’; σ))

tal que el programa de referencia tiene la declaración procedure proc(val y ; res z): S. S[y | y’][z | z’] se obtiene de S renombrando todas las ocurrencias de y con y’, y de z con z’ (se utilizan y’, z’, en lugar de y, z, para evitar conflictos de nombres de variables con el entorno de invocación). La transición captura la idea de la invocación con parámetros: se expande el estado corriente, se asigna el parámetro real por valor al parámetro formal correspondiente, se ejecuta el cuerpo del procedimiento, se devuelve el parámetro formal por resultado al parámetro real correspondiente, y finalmente el estado corriente vuelve a su situación original con la instrucción release, que se especifica con la transición

$$(\text{release}(y', z'), \sigma) \rightarrow (E, \text{exclude}(y', z' ; \sigma))$$

Para la verificación de un procedimiento con parámetros se suele plantear una combinación de dos reglas, una para derivar una propiedad general del procedimiento y otra para adaptarla en términos de los parámetros reales de su invocación. La forma de la primera regla, basada en la regla INVOC que describimos previamente, es

$$\text{Regla PINVOC} \qquad \frac{\{p'\}\ S\ \{q'\}}{\{p'\}\ \text{call proc}(y ; z)\ \{q'\}}$$

tal que el programa de referencia incluye la declaración procedure proc(val y ; res z): S, $\text{free}(p') \subseteq \{y, z\}$, y $\text{free}(q') \subseteq \{z\}$. La regla establece que a partir de una fórmula de correctitud parcial sobre el cuerpo de un procedimiento, se puede inferir la misma fórmula sobre la invocación ficticia al procedimiento, tomando como parámetros reales los formales. Por la semántica del parámetro por valor, la postcondicion no lo incluye (para referenciarlo se puede utilizar una variable de especificación). La segunda regla se basa en la siguiente idea. Dada la fórmula {p'} S {q'}, supongamos que se quiere probar {p} call proc(e ; x) {q}, siendo S el cuerpo de proc, expresado en términos de y, z. Por la asignación inicial del parámetro por valor, debe cumplirse naturalmente $p \rightarrow p'[y \mid e]$. De la misma manera, por la devolución del parámetro por resultado, la postcondición q' debe implicar una apropiada postcondición q. Suponiendo que el efecto de S a partir de *e* es la asignación de un determinado valor *a* al parámetro z, entonces la misma asignación se hace al parámetro x, lo que debe reflejarse en la postcondición q. Así, se debe agregar la aserción $\forall a: (q'[z \mid a] \rightarrow q[x \mid a])$ como precondición de la invocación al procedimiento. La regla completa, conocida como ADAPT, se formula así:

$$\text{Regla ADAPT} \qquad \frac{\{p'\}\ \text{call proc}(y ; z)\ \{q'\}}{\{p'[y \mid e] \wedge \forall a: (q'[z \mid a] \rightarrow q[x \mid a])\}\ \text{call proc}(e ; x)\ \{q\}}$$

Ejemplo 15.4. Correctitud parcial con un procedimiento con parámetros

Empleando las reglas PINVOC y ADAPT, vamos a probar lo siguiente:

$\{u = 0\}$
P_4 :: procedure suma1(val y ; res z): z := y + 1 ;
call suma1(u ; v)
$\{v^2 = v\}$

Los pasos de la prueba son

1. $\{y = Y\}$ z := y + 1 $\{z = Y + 1\}$ (ASI, CONS)
2. $\{y = Y\}$ call suma1(y ; z) $\{z = Y + 1\}$ (1, PINVOC)
3. $\{y = 0\}$ call suma1(y ; z) $\{z = 1\}$ (2, INST, CONS)
4. $\forall a: (a = 1 \rightarrow a^2 = a)$ (MAT)
5. $\{u = 0\}$ call suma1(u ; v) $\{v^2 = v\}$ (3, 4, ADAPT, CONS)

En el último paso se elimina $\forall a: (a = 1 \rightarrow a^2 = a)$, porque es verdadera.

Fin de Ejemplo

Adaptando la regla REC para considerar procedimientos recursivos con parámetros, se formula la regla PREC que se presenta a continuación (seguimos asumiendo un parámetro de cada tipo, y se tienen en cuenta las restricciones previas):

Regla PREC $\dfrac{\{p\}\ \text{call proc(y ; z)}\ \{q\} \vdash \{p\}\ S\ \{q\}}{\{p\}\ \text{call proc(y ; z)}\ \{q\}}$

Con PREC se prueba sin problemas $\{a = A \geq 0\}$ call fact(a ; b) $\{b = A!\}$, siendo fact el siguiente procedimiento que calcula el factorial (la prueba queda como ejercicio):

procedure fact(val y ; res z): if y = 0 then z := 1
else call fact(y – 1 ; z) ; z := y . z fi

TEMA 15.4. VERIFICACIÓN DE PROGRAMAS CONCURRENTES

En esta última sección describimos aspectos relevantes de la verificación de programas concurrentes. Por su complejidad, para este paradigma más que para ningún otro es necesaria una metodología de prueba rigurosa. Para simplificar, nos enfocamos solamente en un lenguaje concurrente con *variables compartidas*.
La forma de un programa concurrente es

$$P :: S_0 ; [S_1 \parallel S_2 \parallel \ldots \parallel S_n]$$

S_0 es un subprograma de PLW, para inicialización de variables. Le sigue una composición concurrente de subprogramas secuenciales o *procesos* S_i, que se puede abreviar $[\parallel_{i=1,n} S_i]$. No hay anidamiento de concurrencia ni creación dinámica de procesos. Un proceso S_i tiene instrucciones de PLW más una instrucción de *retardo condicional*, el await, que es la *primitiva de sincronización*. La sintaxis del await es

await B then S end

tal que B es una expresión booleana y el cuerpo S tiene sólo instrucciones de PLW a excepción del while (es decir que no se permite el anidamiento de instrucciones await, y además S termina siempre). Un proceso ejecuta un await para producir un retardo que le permita avanzar cuando se cumpla un determinado criterio de consistencia asociado a una o más variables compartidas, o bien para obtener exclusividad sobre una *sección crítica*, es decir una sección de programa con variables compartidas modificables. La semántica informal del await es la siguiente. El proceso asociado accede con exclusividad a las variables de la expresión booleana B, y si B resulta verdadera, se ejecuta el cuerpo S completo, sin interrupciones. De esta manera el await se ejecuta atómicamente. Si en cambio B es falsa, se libera el acceso a las variables de la expresión, y el proceso queda bloqueado hasta una etapa posterior, cuando en la misma situación pueda avanzar por una evaluación positiva de B.
Se sigue asumiendo que el skip, la asignación y la evaluación de una expresión booleana son atómicos, para facilitar la metodología de prueba. La comunicación entre los procesos se efectúa a través de las variables compartidas. El control de un programa se sitúa en distintos lugares al mismo tiempo, uno en cada proceso. 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 que se define más habitualmente en concurrencia, de intercalación (*interleaving*) no determinística de las transiciones de los distintos procesos. Un mismo programa puede producir computaciones que terminan, computaciones que fallan (computaciones con *deadlock* o bloqueo mortal, producto del uso de la instrucción await, que se manifiesta con uno o más procesos bloqueados indefinidamente a la espera de un evento que nunca se producirá), y computaciones que no terminan, por el uso de la instrucción while. No hay en principio ninguna hipótesis de *fairness*, la única asunción es la *propiedad de progreso fundamental*: si un proceso puede avanzar, el programa avanza.
La semántica formal de la instrucción await se define de la siguiente manera:

$$\text{Si } \sigma \models B \text{ y } (S, \sigma) \rightarrow^* (E, \sigma'), \text{ entonces (await B then S end, } \sigma) \rightarrow (E, \sigma')$$

La definición refleja la atomicidad de la instrucción. El await siempre termina porque el cuerpo S no tiene instrucciones while. Sólo se avanza si la expresión booleana B resulta verdadera.
Para completar la definición de la semántica formal del lenguaje, resta definir mediante la relación $\rightarrow$ la transición entre las configuraciones concurrentes. Estas tienen la forma $([\parallel_{i=1,n} S_i], \sigma)$, siendo las S_i continuaciones sintácticas de los procesos respectivos, y σ un estado global único con variables de todos los procesos. Se define

$$\text{si } (S_i, \sigma) \rightarrow (S'_i, \sigma'), \text{ entonces } ([S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_n], \sigma) \rightarrow_{(i,\sigma)} ([S_1 \parallel \ldots \parallel S'_i \parallel \ldots \parallel S_n], \sigma')$$

La expresión $\rightarrow_{(i,\sigma)}$ indica que la computación avanza por el proceso i-ésimo, a partir del estado corriente σ. Más en general, se define

$$([\parallel_{i=1,n} S_i], \sigma) \rightarrow^*_h ([\parallel_{i=1,n} S'_i], \sigma')$$

donde $h = (i_1,\sigma)...(i_k,\sigma_k)$, con $\sigma_k = \sigma'$, es la *historia* de las transiciones desde $([\parallel_{i=1,n} S_i], \sigma)$ hasta $([\parallel_{i=1,n} S'_i], \sigma')$. Si una computación $\pi(P, \sigma)$ termina, la configuración terminal tiene la forma $([\parallel_{i=1,n} E_i], \sigma')$, para algún estado σ'.
Consideremos el siguiente ejemplo. El programa P_{cfac} calcula concurrentemente con dos procesos 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 then

if i + 1 < k then i := i + 1 ; n := n . i else c_1 := false fi end od

||

while c_2 do await true then

if k – 1 > i then k := k – 1 ; n := n . k else c_2 := false fi end od]

El primer proceso contribuye con la multiplicación de los primeros números naturales, hasta algún i, y el segundo proceso con la multiplicación de los números siguientes, hasta N. En este caso los await se utilizan solamente para lograr la exclusión mutua.

Dado un estado inicial σ que satisface p, la correctitud parcial de un programa concurrente con respecto a (p, q) se cumple si a partir de σ, todas las computaciones que terminan lo hacen en un estado que satisface q. La correctitud total involucra la correctitud parcial, la terminación y la ausencia de *deadlock* en todas las computaciones. Cuando no se tiene en cuenta el *deadlock* se habla de correctitud total *débil*. Se pueden considerar también otras propiedades, como la exclusión mutua y la ausencia de inanición.

Para la prueba de correctitud parcial, además de los axiomas y reglas de H se deben utilizar naturalmente dos nuevas reglas (en principio), una para el await y la otra para la composición concurrente. En el caso del await, la regla habitual es

$$\text{Regla AWAIT} \qquad \frac{\{p \wedge B\}\ S\ \{q\}}{\{p\}\ \text{await B then S end}\ \{q\}}$$

Como se observa, la forma del AWAIT es muy similar a la de la regla COND para la selección condicional. El eventual bloqueo que puede provocarse por una evaluación negativa de la expresión booleana B se considera en la verificación de la correctitud total. Se demuestra fácilmente que la regla AWAIT es sensata (la prueba queda como ejercicio).

Con respecto a la composición concurrente lo natural es definir, asumiendo composicionalidad, una regla de prueba estableciendo que a partir de las fórmulas de correctitud parcial $\{p_i\}\ S_i\ \{q_i\}$ correspondientes a los procesos S_i, se deriva la fórmula

$$\{\bigwedge_{i=1,n} p_i\} \; [\; \|_{i=1,n} S_i \;] \; \{\bigwedge_{i=1,n} q_i\}$$

Lamentablemente, a diferencia de lo que sucede en la programación secuencial, la composicionalidad se pierde en la concurrencia. Por ejemplo, los procesos

$S_1 :: x := x + 2$
$S_2 :: x := x + 1 \; ; \; x := x + 1$

son funcionalmente equivalentes, o en otras palabras, cumplen para todo par (p, q):

$$\models \{p\} \; x := x + 2 \; \{q\} \;\; \leftrightarrow \;\; \models \{p\} \; x := x + 1 \; ; \; x := x + 1 \; \{q\}$$

pero compuestos concurrentemente con el proceso

$T :: x := 0$

producen programas con comportamientos distintos:

$\models \{true\} \; [x := x + 2 \; \| \; x := 0] \; \{x = 0 \vee x = 2\}$, considerando S_1 y T
$\models \{true\} \; [x := x + 1 \; ; \; x := x + 1 \; \| \; x := 0] \; \{x = 0 \vee x = 1 \vee x = 2\}$, considerando S_2 y T

Es decir, los procesos funcionamente equivalentes S_1 y S_2 no son intercambiables en una composición concurrente, no pueden tratarse como *cajas negras* como se hace en la programación secuencial: ahora debe analizarse cómo la computación de un proceso se ve afectada por el resto. En este caso, resulta imprescindible recurrir a las *proof outlines* estándar que definimos en la Clase 12 (de ahora en más las llamaremos directamente *proof outlines* para abreviar la nomenclatura). Se plantea una prueba en dos etapas. En una primera etapa se construye una *proof outline* para cada proceso. Y en la segunda etapa las *proof outlines* se consisten: se chequea que todas las aserciones intermedias sean verdaderas cualquiera sea el *interleaving* entre las transiciones de los distintos procesos (se dice que las *proof outlines* deben ser *libres de interferencia*). Así, a partir de la precondición $\bigwedge_{i=1,n} p_i$, luego de la ejecución del programa $[\; \|_{i=1,n} S_i \;]$ valdrá naturalmente la postcondición $\bigwedge_{i=1,n} q_i$. La preservación composicional en las *proof*

outlines secuenciales, producto de la sensatez fuerte del método H extendido con la regla AWAIT, no asegura que las aserciones intermedias de la *proof outline* concurrente obtenida componiendo las *proof outlines* secuenciales se mantegan verdaderas (propiedad que se conoce como *preservación concurrente*). Ya lo vimos recién, el contenido de una variable compartida en un momento dado depende de las asignaciones que se le hayan efectuado desde distintos procesos. Por ejemplo, a partir de las *proof outlines*

$\{x = 0\}\ x := x + 2\ \{x = 2\}$
$\{x = 0\}\ x := 0\ \{true\}$

no es correcto formular como *proof outline* concurrente

$$\{x = 0 \wedge x = 0\}\ [x := x + 2 \parallel x := 0]\ \{x = 2 \wedge true\}$$

porque al finalizar el programa, el valor de x puede ser 0. Para chequear la preservación concurrente, o lo que es lo mismo, que las *proof outlines* sean libres de interferencia, el mecanismo utilizado es el siguiente: dada una aserción r en una *proof outline* y una aserción pre(T) en otra, tal que T es un await o una asignación no incluida en un await, tiene que valer $\{r \wedge pre(T)\}\ T\ \{r\}$. Es decir, se fuerza la preservación concurrente asegurando que toda aserción se mantenga verdadera luego de la ejecución de cualquier instrucción atómica de otro proceso que modifique variables compartidas. En la práctica, como veremos, esto se logra debilitando las aserciones originales.
Así llegamos a la formulación de la regla de prueba habitual para la composición concurrente. Como en otros casos, se plantea una meta-regla, dado que la premisa no se expresa en términos de fórmulas de correctitud:

Regla CONC $\dfrac{\{p_i\}\ S_i\ \{q_i\},\ 1 \le i \le n,\ \text{son } \textit{proof outlines} \text{ libres de interferencia}}{\{\bigwedge_{i=1,n} p_i\}\ [\ \|_{i=1,n} S_i\ ]\ \{\bigwedge_{i=1,n} q_i\}}$

Aunque no se cumple la composicionalidad, se establece de todos modos una aproximación composicional basada en una verificación en dos etapas, que mantiene la idea de probar un programa a partir de las pruebas de sus componentes. El chequeo de

que las *proof outlines* sean libres de interferencia tornan las pruebas de los programas concurrentes bastante más trabajosas que las de los programas secuenciales (dados dos procesos de tamaño n_1 y n_2, respectivamente, en la segunda etapa hay que verificar $n_1.n_2$ fórmulas de correctitud). De todas maneras, en la práctica muchas validaciones se resuelven trivialmente. El caso más típico se da cuando la aserción r y la instrucción T no comparten variables. Otro caso de simplificación se cumple cuando la conjunción $r \wedge pre(T)$ es falsa, situación que representa un *interleaving* que no puede suceder. Se demuestra fácilmente que la regla CONC es sensata (la prueba queda como ejercicio).
Antes de presentar un ejemplo de prueba en este marco, tenemos que completar la definición inductiva de las proof outlines desarrollada en la Clase 12, agregando el await. Dada la instrucción S :: await B then T end, se define:

$$pre(S) \wedge B \rightarrow pre(T), \text{ y } post(T) \rightarrow post(S)$$

Ejemplo 15.5. Correctitud parcial de programas concurrentes

Se quiere probar $\{x = 0\}\ [x := x + 1 \parallel x := x + 2]\ \{x = 3\}$. Proponemos las siguientes *proof outlines*:

$\{x = 0 \vee x = 2\}\ x := x + 1\ \{x = 1 \vee x = 3\}$
$\{x = 0 \vee x = 1\}\ x := x + 2\ \{x = 2 \vee x = 3\}$

Se cumple que las *proof outlines* son libres de interferencia, es decir que las siguientes fórmulas son verdaderas:

$\{(x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)\}\ x := x + 2\ \{x = 0 \vee x = 2\}$
$\{(x = 1 \vee x = 3) \wedge (x = 0 \vee x = 1)\}\ x := x + 2\ \{x = 1 \vee x = 3\}$
$\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2)\}\ x := x + 1\ \{x = 0 \vee x = 1\}$
$\{(x = 2 \vee x = 3) \wedge (x = 0 \vee x = 2)\}\ x := x + 1\ \{x = 2 \vee x = 3\}$

También vale:

$x = 0 \rightarrow ((x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1))$
$((x = 1 \vee x = 3) \wedge (x = 2 \vee x = 3)) \rightarrow x = 3$

De esta manera, se cumple $\{x = 0\}\ [x := x + 1 \parallel x := x + 2]\ \{x = 3\}$.

Volviendo, por otra parte, al programa P_{cfac} que calcula el factorial, se prueba $\{N > 1\}\ P_{cfac}\ \{n = N!\}$.

Se proponen las siguientes *proof outlines*:

$\{inv: i < k \wedge (\neg c_1 \rightarrow i + 1 = k) \wedge n.(i + 1)\ldots(k - 1) = N!\}$

while c_1 do $\{c_1 \wedge i < k \wedge n.(i + 1)\ldots(k - 1) = N!\}$

await true then if $i + 1 < k$ then $i := i + 1$; $n := n\ .\ i$ else $c_1 := false$ fi end

$\{i < k \wedge (\neg c_1 \rightarrow i + 1 = k) \wedge n.(i + 1)\ldots(k - 1) = N!\}$ od

$\{i < k \wedge i + 1 = k \wedge n.(i + 1)\ldots(k - 1) = N!\}$

$\{inv: i < k \wedge (\neg c_2 \rightarrow k - 1 = i) \wedge n.(i + 1)\ldots(k - 1) = N!\}$

while c_2 do $\{c_2 \wedge i < k \wedge n.(i + 1)\ldots(k - 1) = N!\}$

await true $\rightarrow$ if $k - 1 > i$ then $k := k - 1$; $n := n\ .\ k$ else $c_2 := false$ fi end

$\{i < k \wedge (\neg c_2 \rightarrow k - 1 = i) \wedge n.(i + 1)\ldots(k - 1) = N!\}$ od

$\{i < k \wedge k - 1 = i \wedge n.(i + 1)\ldots(k - 1) = N!\}$

Notar que no se muestran las aserciones internas de los await. En la primera etapa deben efectivamente encontrarse, pero en la segunda no se consideran porque los await son atómicos. A partir de estas *proof outlines*, se llega sin problemas a $\{N > 1\}\ P_{cfac}\ \{n = N!\}$. Queda como ejercicio completar la prueba.

Fin de Ejemplo

El debilitamiento de las aserciones, necesario para que las *proof outlines* sean libres de interferencia, atenta contra la completitud del método de verificación. Por ejemplo, se cumple

$$\models \{x = 0\}\ [x := x + 1 \parallel x := x + 1]\ \{x = 2\}$$

Sin embargo, dicha fórmula no puede demostrarse en H extendido con AWAIT y CONC (no lo probaremos). *Proof outlines* candidatas serían

$\{x = 0 \vee x = 1\}\ x := x + 1\ \{x = 1 \vee x = 2\}$

$\{x = 0 \vee x = 1\}\ x := x + 1\ \{x = 1 \vee x = 2\}$

pero claramente no satisfacen los requisitos de la regla CONC. A diferencia del programa similar visto recién, en este caso las aserciones intermedias son demasiado débiles. Se avance primero por un proceso o por el otro, el estado intermedio σ es el mismo ($\sigma \models x = 1$), y solamente con la variable de programa x no alcanza para registrar la historia del *interleaving* que se lleva a cabo.

Para restituir la completitud se suele incorporar una nueva regla de prueba. La regla se basa en la ampliación del programa original con *variables auxiliares*, para justamente fortalecer las aserciones intermedias pero sin afectar el cómputo básico. Considerando el ejemplo anterior podría hacerse lo siguiente:

- Agregar dos variables auxiliares, la variable *y* en el primer proceso, y la variable z en el segundo, inicializadas en 0.
- Hacer y := 1 después que el primer proceso ejecuta la asignación x := x + 1, y hacer lo mismo con z en el segundo proceso.
- Fortalecer las aserciones con la información de las variables auxiliares.

De esta manera, el inicio del programa ampliado cumple

$\{x = 0\}\ y := 0\ ;\ z := 0\ ;\ \{x = 0 \wedge y = 0 \wedge z = 0\}$

y las nuevas *proof outlines* son:

$\{(x = 0 \wedge y = 0 \wedge z = 0) \vee (x = 1 \wedge y = 0 \wedge z = 1)\}$
await true then x := x + 1 ; y := 1 end
$\{(x = 1 \wedge y = 1 \wedge z = 0) \vee (x = 2 \wedge y = 1 \wedge z = 1)\}$

$\{(x = 0 \wedge y = 0 \wedge z = 0) \vee (x = 1 \wedge y = 1 \wedge z = 0)\}$
await true then x := x + 1 ; z := 1 end
$\{(x = 1 \wedge y = 0 \wedge z = 1) \vee (x = 2 \wedge y = 1 \wedge z = 1)\}$

Ahora se cumplen los requisitos de la regla CONC, y así se prueba $\{x = 0\}$ P' $\{x = 2\}$, siendo P' el programa P ampliado con las variables auxiliares. Pero por el tipo de modificaciones llevadas a cabo, la prueba también aplica al programa original P.
Formalizando, se define que A es un conjunto de variables auxiliares con respecto a un programa P y a aserciones p y q, si cumple

- $A \cap free(q) = \varnothing$.
- Las variables de A sólo aparecen en asignaciones de P.
- Toda asignación x := e de P cumple que si la expresión *e* incluye una variable de A, entonces x está en A.

Notar que puede ser $A \cap free(p) \neq \varnothing$ (dichas variables se tratan como variables de especificación). Con estas consideraciones, presentamos la nueva regla de prueba:

Regla AUX $$\frac{\{p\}\ P\ \{q\}}{\{p\}\ P_{|A}\ \{q\}}$$

tal que $P_{|A}$ es el programa P sin las nuevas asignaciones con variables auxiliares. Se demuestra fácilmente que la regla AUX es sensata (la prueba queda como ejercicio).
La sensatez fuerte y la completitud del método de verificación de correctitud parcial son base para la prueba de las otras propiedades de un programa concurrente. En efecto, la metodología que estamos describiendo plantea un esquema de prueba en dos etapas para todas las propiedades. En la primera etapa se encuentran *proof outlines* para cada proceso, las cuales pueden variar de prueba en prueba. Y en la segunda etapa las mismas se consisten, con un criterio que depende de la propiedad.
Por ejemplo, en el caso de la propiedad de terminación, la segunda etapa consiste en verificar que las *proof outlines* son libres de interferencia, pero además que para todo invariante parametrizado p(w) de una proof outline y toda aserción pre(T) de otra, siendo T un await o una asignación fuera de un await, se cumple $\langle p(w) \wedge pre(T)\rangle\ T\ \langle \exists v \leq w: p(v)\rangle$. Es decir, no basta con probar aisladamente que todo while termina, sino que también debe verificarse que ninguna instrucción de otro proceso que puede modificar variables compartidas perjudique la terminación del while incrementando el variante asociado (puede en cambio acelerar la terminación si lo decrementa). Queda como

ejercicio probar la terminación del programa P_{cfac} del factorial utilizando este mecanismo. En particular, con hipótesis de *fairness* la prueba de terminación contempla la posibilidad de instrucciones while que, consideradas aisladamente, pueden no terminar. Por ejemplo, dado el programa

P :: [while x = 1 do skip od || x := 0]

si la hipótesis de *fairness* asegura que la asignación x := 0 se va a ejecutar alguna vez, entonces P termina aún cuando el while, considerado aisladamente, no termina.
En el caso de la propiedad de ausencia de *deadlock*, la segunda etapa consiste en:

1. Marcar en la *proof outline* concurrente todos los casos posibles de *deadlock*. Esto se hace identificando n-tuplas $C_i = (\lambda_1, \ldots, \lambda_n)$, tantas como casos posibles de *deadlock* existan, tal que el programa tiene n procesos, y cada λ_k es una etiqueta asociada a un await o al final de un proceso. Toda C_i debe tener al menos una etiqueta asociada a un await, porque de lo contrario no representaría un caso posible de *deadlock*.
2. Caracterizar semánticamente las n-tuplas C_i mediante aserciones δ_i, y probar que todas son falsas. Las aserciones δ_i se conocen como *imágenes semánticas*. Son conjunciones de aserciones δ_{ik} que cumplen:
   2.1. Si la etiqueta λ_k de C_i está asociada a una instrucción T :: await B then S end, entonces $\delta_{ik} = pre(T) \wedge \neg B$.
   2.2. Si la etiqueta λ_k de C_i está asociada al final de un proceso S_k, entonces $\delta_{ik} = post(S_k)$.

Supongamos, por ejemplo, la siguiente estructura de una *proof outline* concurrente de {p} P {q}, en la que ya aparecen las etiquetas λ_k:

P :: [S_1 :: … {pre(T_1)} $\lambda_1 \rightarrow T_1$ :: await B_1 then S_1 end … $\lambda_2 \rightarrow$ {post(P_1)}
||
S_2 :: … $\lambda_3 \rightarrow$ {post(P_2)}
||
S_3 :: … {pre(T_2)} $\lambda_4 \rightarrow T_2$ :: await B_2 then S_2 end … $\lambda_5 \rightarrow$ {post(P_3)}]

Los casos posibles de *deadlock* son

$C_1 = <\lambda_1, \lambda_3, \lambda_4>$
$C_2 = <\lambda_1, \lambda_3, \lambda_5>$
$C_3 = <\lambda_2, \lambda_3, \lambda_4>$

Y las imágenes semánticas asociadas a las ternas C_i son

$\delta_1 = (pre(T_1) \wedge \neg B_1) \wedge post(P_2) \wedge (pre(T_2) \wedge \neg B_2)$
$\delta_2 = (pre(T_1) \wedge \neg B_1) \wedge post(P_2) \wedge post(P_3)$
$\delta_3 = post(P_1) \wedge post(P_2) \wedge (pre(T_2) \wedge \neg B_2)$

Así, verificando que las aserciones δ_1, δ_2 y δ_3 son falsas, se prueba que P, a partir de un estado inicial que satisface la precondición p, no tiene *deadlock*. Queda como ejercicio probar que el programa P_{cfac} del factorial no tiene *deadlock* en base a este mecanismo.
Utilizando una primitiva de sincronización que fuerza automáticamente la exclusión mutua de las secciones críticas, la verificación de un programa concurrente se torna más simple. Asumamos en lo que sigue que el lenguaje de programación tiene una primitiva de este tipo, de la forma

with r_k when B do S endwith

tal que r_k es un *recurso*, B una expresión booleana, y S un subprograma de PLW (para simplificar, no se permite el anidamiento de instrucciones with). Los recursos son conjuntos disjuntos de variables. Las variables de los recursos sólo pueden ser utilizadas por las sentencias with, y toda variable compartida modificable debe estar definida dentro de un recurso (de esta manera se pueden identificar claramente las secciones críticas de un programa).
La semántica informal del with es la siguiente. Cuando un proceso S_i está por ejecutar una instrucción with r_k when B do S endwith, si el recurso está libre y B es verdadera, entonces S_i puede ocupar el recurso, obtener el uso exclusivo del mismo y progresar en la ejecución del with. Recién cuando S_i completa el with libera el recurso, para que algún otro proceso lo utilice. No hay manejo de prioridades sobre los recursos. Los with no son atómicos, y naturalmente pueden causar *deadlock*.

El mecanismo descripto es una simplificación de los *monitores*. Un monitor es más sofisticado, permite establecer políticas de priorización de procesos, liberaciones temporarias, y la implementación de *tipos de datos abstractos*, encapsulando datos y operaciones.

Una regla de prueba habitual para la instrucción with, asumiendo un solo recurso r, es

$$\text{Regla WITH} \qquad \frac{\{I_r \wedge p \wedge B\}\ S\ \{I_r \wedge q\}}{\{p\}\ \text{with r when B do S endwith}\ \{q\}}$$

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 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. Correspondientemente, para la composición concurrente se utiliza la meta-regla SCC (por *sección crítica condicional*):

$$\text{Regla SCC} \qquad \frac{\{p_i\}\ S_i\ \{q_i\},\ 1 \le i \le n, \text{ son } \textit{proof outlines} \text{ que utilizan } I_r}{\{I_r \wedge \bigwedge_{i=1,n} p_i\}\ [\ \|_{i=1,n} S_i\ ]\ \{I_r \wedge \bigwedge_{i=1,n} q_i\}}$$

tal que para todo $i \neq k$, $(\text{free}(p_i) \cup \text{free}(q_i)) \cap \text{change}(S_k) = \emptyset$, y $\text{free}(I_r) \subseteq r$.

Ahora, en la segunda etapa de la prueba no se necesita chequear que las *proof outlines* sean libres de interferencia, porque sus aserciones se refieren únicamente a variables no compartidas 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 son más estructuradas que las que vimos anteriormente.

Ejercicios de la Clase 15

1. Una aproximación conocida para la definición semántica de los arreglos consiste en diferenciar la parte izquierda de la parte derecha de una variable suscripta. Dada una variable a[e], se define $L(a[e])(\sigma) = (a, R(e)(\sigma))$ para la parte izquierda y $R(a[e])(\sigma) = \sigma(L(a[e])(\sigma))$ para la parte derecha (L por *left*, izquierda, y R por *right*, derecha).

Es decir, para determinar el valor de una variable a[e], se le asocia a a[e] un par (a, i), siendo i el valor de e en el estado corriente σ (parte izquierda), y se aplica σ sobre (a, i) para obtener su valor (parte derecha). Por ejemplo, una asignación $a[e_1] := b[e_2]$, dado un estado σ, se interpreta de la siguiente manera: primero se evalúa e_1 en σ y se obtiene i, y se le asocia a $a[e_1]$ el par (a, i); luego se evalúa e_2 en σ y se obtiene k, y se le asocia a $b[e_2]$ el par (b, k); luego se aplica σ sobre (b, k) y se obtiene n; y finalmente se reemplaza σ por σ', que es igual a σ salvo que ahora σ'((a, i)) = n. Inductivamente, se define entre otras cosas:

1. $L(x)(\sigma) = x$
2. $L(a[e])(\sigma) = (a, R(e)(\sigma))$
3. $R(n)(\sigma) = n$, siendo el n de la izquierda un numeral y el n de la derecha un entero
4. $R(x)(\sigma) = \sigma(L(x)(\sigma))$
5. $R(e_1 + e_2)(\sigma) = R(e_1)(\sigma) + R(e_2)(\sigma)$

Se pide:

i. Completar las definiciones que faltan. Fundamentalmente se apunta a llegar a definir la semántica de la asignación considerando arreglos.

ii. Probar, usando el inciso i, que se cumple $M(x := x + 1)(\sigma[x \mid 0]) = \sigma[x \mid 1]$.

iii. Probar, también usando el inciso i, que se cumple $M(a[a[2]] := 1)(\sigma[(a, 1) \mid 2][(a, 2) \mid 2]) = \sigma[(a, 1) \mid 2][(a, 2) \mid 1]$.

2. Sea *a* un arreglo de valores enteros a[0:N – 1], con N > 0. Se define que una sección de *a* es un fragmento a[i:k], con $0 \leq i \leq k < N$. La expresión $s_{i,k}$ denota la suma de a[i:k], es decir $\Sigma_{n=i,k}\ a[n]$. Una sección de mínima suma, o directamente una sección mínima, de a[0:N – 1], es una sección a[i:k] tal que su suma es mínima considerando todas las secciones de a. Por ejemplo, la sección mínima de a[0:4] = (5, –3, 2, –4, 1) es a[1:3], cuya suma es –5, y las secciones mínimas de a[0:4] = (5, 2, 5, 4, 2) son a[1:1] y a[4:4], con suma 2. Se pide construir sistemáticamente un programa S_{minsum} que devuelva en una variable x la suma de la sección mínima de una arreglo a[0:N – 1], es decir que satisfaga $\langle N > 0 \rangle\ S_{minsum}\ \langle x = \min\{s_{i,k} \mid 0 \leq i \leq k < N\}\rangle$. Se establece que $a \notin change(S_{minsum})$.
3. Probar que la sensatez y la completitud del método H se preservan, cuando H se extiende con la regla INVOC para verificar programas PLW con procedimientos.
4. Completar la prueba del Ejemplo 15.2.
5. Completar las pruebas del Ejemplo 15.3.
6. Probar, utilizando el método H extendido para programas con procedimientos:

i. $\{n = N \geq 0 \wedge s = 0\}$

procedure p:

if n = 0 then skip

else n := n – 1; call p ; s := s + 1; call p ; n := n + 1 fi ;

call p

$\{n = N \wedge s = 2^N - 1\}$

ii. $\langle x = N \geq 0 \rangle$

procedure d:

if x = 0 then skip else x := x – 1 ; call d ; x := x + 2 fi ;

call d

$\langle x = 2N \rangle$

iii. $\{a = N \geq 0\}$

procedure incdec(val i ; res x):

while i > 0 do x := x + 1 ; i := i – 1 od ;

call incdec (a, a)

$\{a = 2N\}$

iv. $\{a = A \geq 0\}$

procedure fact(val y ; res z):

if y = 0 then z := 1 else call fact(y – 1 ; z) ; z := y . z fi ;

call fact(a ; b)

$\{b = A!\}$

7. Probar la sensatez de las reglas AWAIT, CONC y AUX, utilizadas para verificar programas concurrentes con la primitiva de sincronización await.
8. Probar la sensatez de una variante de la regla CONC, en la que no se exige que las *proof outlines* sean libres de interferencia, siempre que los procesos no compartan variables modificables.
9. Completar la prueba del Ejemplo 15.5.
10. Probar:
    i. $\{true\}\ [x := 1 ; x := 2 \parallel x := x + 1]\ \{x = 2 \vee x = 3\}$
    ii. $\{true\}\ [x := x + 1 ; x := x + 1 \parallel x := 0]\ \{x = 0 \vee x = 1 \vee x = 2\}$
11. Mostrar si se puede probar la fórmula siguiente sin recurrir a la regla AUX:

    $\{true\}\ [x := 0 \parallel x := x + 1]\ \{x = 0 \vee x = 1\}$

12. Probar la sensatez de las reglas de terminación y ausencia de *deadlock* propuestas en la Clase 15, en el marco de los programas concurrentes con la instrucción await.
13. Probar la terminación y la ausencia de *deadlock* del programa P_{cfac} del factorial mostrado en la Clase 15, a partir de la precondición $N > 1$.
14. Las operaciones p y v de un semáforo s pueden definirse, empleando la instrucción await, de la siguiente manera:

    p(s) :: await s > 0 then s := s – 1 end

    v(s) :: await true then s := s + 1 end

    Probar que en el siguiente fragmento de programa, que emplea las operaciones p y v para lograr exclusión mutua, no hay *deadlock*:

    SC_{await} :: s := 1; [S_1 || S_2], con:

    S_i :: while true do <sección no crítica> ; p(s) ; <sección crítica> ; v(s) od

    Asumir que en las secciones referidas no aparece la variable s.
15. Sea el siguiente programa:

    P :: c := true ; p := false ;

    [while true do await c then c := false end ; A_1 ; await ¬p then p := true end ;

    await c then c := false end ; B_1 ; await ¬p then p := true end od

    ||

    while true do await p then p := false end ; A_2 ; await ¬c then c := true end ;

    await p then p := false end ; B_2 ; await ¬c then c := true end od]

    A_1 y B_1 son dos secciones críticas del primer proceso, y A_2 y B_2 del segundo proceso. Las mismas no modifican las variables booleanas c y p. Probar que en P se logra la exclusión mutua.
16. Para definir la semántica del lenguaje concurrente que utiliza recursos, planteado al final de la Clase 15, se suele recurrir a un arreglo ρ[k], con $1 \leq k \leq m$, asociado a los m recursos r_k de un programa, tal que ρ[k] = 0 o 1 según r_k está libre u ocupado, respectivamente. Al inicio se cumple que para todo k, ρ[k] = 0. Se define, entre otras cosas, que si $S_i \neq$ with y $(S_i, \sigma) \rightarrow (S'_i, \sigma')$, entonces $([S_1 \| \ldots \| S_i \| \ldots \| S_n], \rho, \sigma) \rightarrow_{(i,\sigma)} ([S_1 \| \ldots \| S'_i \| \ldots \| S_n], \rho, \sigma')$, es decir que las instrucciones que no son with no modifican la situación de los recursos. Se pide completar la definición semántica para el caso en que S_i = with. En otras palabras, teniendo en cuenta que el with no es atómico, se debe definir la relación $\longrightarrow$ a partir de ([S_1 || … || with r_k when B do S_i endwith || … || S_n], ρ, σ) y de ([S_1 || … || endwith || … || S_n], ρ, σ).

17. También al final de la Clase 15 se mencionó al monitor como un mecanismo más sofisticado que el recurso, permitiendo priorizar procesos, liberaciones temporarias y la implementación de tipos de datos abstractos por encapsulamiento de datos y operaciones. En este caso se considera que un recurso r tiene además de variables de programa, variables de condición $r.c_1$, …, $r.c_k$, 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 cuando 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 del lenguaje para contemplar el uso de monitores suele ser la siguiente. Al arreglo ρ asociado a los m recursos se le agregan otros, γ_1, …, γ_m, que contienen conjuntos de índices de procesos. El significado de $\gamma_k[h] = A$ es que para todo $i \in A$ se cumple que P_i es un proceso suspendido por un wait $r_k.c_h$. Además se agrega el arreglo $\gamma_0[1:m]$, 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.
18. Probar la sensatez de las reglas WITH y SCC, utilizadas para verificar programas concurrentes con la primitiva de sincronización with. ¿Por qué en la conclusión de la regla WITH no se agrega, en la pre y postcondición, el invariante I_r del recurso r?
19. Proponer reglas para la prueba de terminación y de ausencia de *deadlock* en los programas con la instrucción with, y probar su sensatez.
20. Considérese la siguiente solución, utilizando la instrucción with, para lograr la exclusión mutua:

    SC_{with} :: resource semaforo(s) ; s := 1 ; [S_1 || S_2], con:

    S_i :: with semáforo when s = 1 do s := 0 endwith ;

        <sección crítica> ;

        with semaforo when true do s := 1 endwith

Mostrar que en SC_{with} no hay *deadlock*.

21. Determinar sobre cuáles de las siguientes propiedades de un programa concurrente impacta el *fairness*: correctitud parcial, terminación, ausencia de *deadlock*, exclusión mutua, ausencia de inanición.

Notas y bibliografía para la Parte 3

Las primeras publicaciones relacionadas con una metodología para la verificación de programas que se registran, se ubican entre comienzos de los años 1950 y la primera mitad de la década de 1960 (trabajos de A. Turing, H. Goldstine, J. von Neumann y P. Naur). Pero la contribución que se considera fundacional fue (Floyd, 1967), en la que mediante métodos axiomáticos se trata la prueba de correctitud parcial con aserciones inductivas y la prueba de terminación con variantes de órdenes parciales bien fundados, utilizando diagramas de flujo. El primer sistema de prueba composicional, con programas secuenciales determinísticos de entrada/salida provistos de la estructura de control while, se presentó en (Hoare, 1969); este trabajo impulsó sobremanera la disciplina completa de la verificación de programas. Para leer sobre la historia de la verificación de programas, ver por ejemplo (Jones, 1992).

En (Plotkin, 1981) se describe la semántica operacional estructural. De las publicaciones existentes sobre la aproximación denotacional para definir la semántica de un lenguaje de programación, recomendamos (de Bakker, 1980); esta obra incluye un apéndice sobre la expresividad de la aritmética de Peano de primer orden. En el marco de la semántica denotacional, en (Loeckx & Sieber, 1987) se desarrolla un enfoque alternativo de la verificación de programas determinísticos al de la lógica de Hoare. Hemos indicado en la Clase 11 que existe también una aproximación axiomática de especificación, utilizando axiomas y reglas de verificación para definir directamente la semántica de los lenguajes de programación; se puede recurrir a (Hoare & Wirth, 1973) para encontrar una definición axiomática de la semántica del lenguaje Pascal. Para leer sobre la especificación de dominios de computación más generales que el que hemos considerado en este libro, y empleando la semántica algebraica (también mencionada en la Clase 11), ver por ejemplo (Ehrig & Mahr, 1985), en que se utilizan especificaciones algebraicas de tipos abstractos de datos. Se pueden plantear teorías de verificación considerando funciones parciales, como por ejemplo la de (Tucker & Zucker, 1988); en este trabajo se estudia la verificación de programas sobre conjuntos de interpretaciones definidas axiomáticamente.

Uno de los primeros intentos de sistema de prueba orientado por la sintaxis de programas que incluyen un uso restringido de la instrucción goto fue (Clint & Hoare, 1972), en el que se desarrollan pruebas con asunciones. En (Francez, 1983) se plantea

una interesante extensión del método H, para deducir aserciones de correctitud parcial de un programa a partir de aserciones de correctitud de otro. El trabajo (Constable & O'Donnel, 1978) presenta programas *anotados*, que se consideran precursores de las *proof outlines* que hemos introducido en la Clase 12. En (Jansen, 1983) se expone una amplia discusión interdisciplinaria sobre la propiedad de composicionalidad, abarcando la computación, las matemáticas y la lingüística.

Nuestra descripción del método H* para la verificación de la terminación de programas sigue fundamentalmente la presentación de (Apt, 1981), inspirada en reglas de verificación conocidas como *reglas de Harel*, basadas en la lógica dinámica. Lecturas recomendadas sobre *fairness* son (Francez, 1986) y (Grumberg, Francez, Makowsky & de Roever, 1985).

El trabajo (Cook, 1978) se considera punto de partida para el estudio de la sensatez y la completitud de los métodos de verificación de programas; en este caso se utiliza un lenguaje de programación más sofisticado que PLW, que incluye procedimientos y considera reglas de alcance, y además la completitud se refiere a una clase de interpretaciones. En (Bergstra & Tucker, 1981) se muestra que la expresividad no es condición necesaria para que se cumpla la completitud. La noción de completitud aritmética se debe a (Harel, 1979). En (Clarke, 1985) se compendian resultados de la completitud de la lógica de Hoare considerando el uso de procedimientos. Los trabajos sobre completitud basados en la semántica denotacional también se desarrollaron desde fines de la década de 1970; ver por ejemplo (de Bakker, 1980), donde se consideran dominios más generales, y se tratan además las excepciones.

La insensatez del axioma de asignación cuando se utilizan arreglos se trató en (de Bakker, 1980), entre otras publicaciones; se describe un sistema completo de reglas de verificación para programas con arreglos.

El desarrollo sistemático de los programas secuenciales, tomando como guía los métodos axiomáticos para las pruebas *a posteriori*, se inició con (Dijkstra, 1976); un programa se va construyendo y probando mediante axiomas y reglas, proceso que termina cuando se obtiene la precondición más débil del programa. Para analizar la construcción de un programa no trivial junto con su prueba de correctitud, ver por ejemplo (Hoare, 1971a).

El primer análisis sobre la verificación de programas con procedimientos apareció en (Hoare, 1971b); se describen reglas de invocación, recursión y adaptación, como las que hemos presentado en nuestro libro. En (Martin, 1983) se discute el relajamiento de las

restricciones para permitir el uso de alias. Sobre otros resultados relacionados con el pasaje de parámetros y el uso de variables locales, recomendamos la lectura de (Apt, 1981) y (Gries & Levin, 1980).

En (Owicki & Gries, 1976a) y (Owicki & Gries, 1976b) se planteó la extensión al método de Hoare para la prueba en dos tiempos de los programas concurrentes con variables compartidas. Los lenguajes con recursos de variables presentados en dichos trabajos evolucionaron hacia el uso de los monitores, para encapsular datos y su manipulación concurrente. La verificación de programas con monitores apareció en (Howard, 1976), entre otras publicaciones. En la tercera parte de nuestro libro anterior (Rosenfeld & Irazábal, 2010) se describe un método de prueba en dos tiempos para los programas distribuidos, publicado inicialmente en (Apt, Francez & de Roever, 1980). El libro (Chandy & Misra, 1988) presenta una sistematización para la derivación de programas concurrentes a partir de especificaciones expresadas en lógica temporal; se lo considera fundacional como lo fue el de E. Dijkstra para la programación secuencial.

El uso de la lógica temporal en la verificación de programas (fuera del alcance de nuestro libro) arrancó con (Pnueli, 1977), en que se consideran los sistemas reactivos. Con la lógica temporal se pueden expresar hipótesis de *fairness*, y también propiedades adicionales a las del comportamiento de entrada/salida de los programas. Para leer más sobre este tema sugerimos (Manna & Pnueli, 1991) y (Manna & Pnueli, 1995). En el caso especial de los programas que operan sólo con estados finitos, la verificación automática es posible. (Queille & Sifakis, 1981) y (Emerson & Clarke, 1982) iniciaron el desarrollo de herramientas para chequear automáticamente si tales programas satisfacen especificaciones escritas en lenguajes de aserciones basados en la lógica temporal; en términos lógicos, se verifica que los programas sean modelos de las especificaciones, y por eso esta aproximación se conoce como *model checking*.

Para pasar revista al estado del arte del soporte herramental a los métodos formales (no sólo de verificación de programas), recomendamos (Woodcock, Gorm Larsen, Bicarregui & Fitzgerald, 2009), reciente presentación muy completa que describe la aplicación de dichos métodos en la industria. En particular, destacamos las referencias a los lenguajes de especificación JML y Spec#, que incluyen el manejo de pre y postcondiciones e invariantes, con soporte herramental para verificación estática y chequeo en tiempo de ejecución.

Para profundizar en los distintos temas sobre verificación de programas presentados en las cinco clases de la tercera parte del libro, incluyendo referencias históricas y

bibliográficas, se recomienda recurrir a (Apt & Olderog, 1997) y (Francez, 1992). Sugerimos para consulta (Apt, 1981), en que se lleva a cabo una muy completa revisión de los métodos de prueba composicional para programas secuenciales determinísticos de entrada/salida, incluyendo el uso de procedimientos con recursión y parámetros. También recomendamos (Backhouse, 1986) y (Baber, 1987), trabajos que describen la verificación de programas secuenciales acentuando consideraciones metodológicas. El primero desarrolla varios ejemplos de pruebas de correctitud parcial de programas PLW en H utilizando arreglos. El segundo tiene una pequeña sección sobre concurrencia. Nuestra última sugerencia es el libro (Huth & Ryan, 2004). En él se clasifica a los métodos de verificación según estén orientados a pruebas o a modelos, y se tratan, respectivamente, en los capítulos 3 y 4. La primera aproximación es la que consideramos en nuestro libro: un sistema se describe mediante un conjunto de fórmulas Γ, de una lógica determinada, y se especifica mediante otro conjunto de fórmulas Φ, de modo tal que la verificación consiste en encontrar una prueba de $\Gamma \vdash \Phi$. En el caso de la aproximación orientada a modelos, en cambio, un sistema se representa por un modelo M en el marco de una apropiada lógica. Una especificación, otra vez, es un conjunto de fórmulas Φ, y la verificación consiste en chequear si M es un modelo o satisface Φ, es decir si $M \models \Phi$. El chequeo se puede realizar automáticamente si el modelo es finito.
Se detallan a continuación las referencias bibliográficas mencionadas previamente:

- Apt, K. (1981). "Ten years of Hoare's logic, a survey, part 1". *ACM Trans. Prog. Lang. Syst.*, 3, 431-483.
- Apt, K., Francez, N. & de Roever, W. (1980). "A proof system for communicating sequential processes". *ACM Trans. Prog. Lang. Syst.*, 2, 359-385.
- Apt, K. & Olderog, E. (1997). *Verification of secuencial and concurrent programs, second edition*. Springer-Verlag.
- Baber, R. (1987). *The spine of software, designing provably correct software – theory and practice.* Wiley.
- Backhouse, R. (1986). *Program construction and verification*. Englewood Cliffs.
- Bergstra, J. & Tucker, J. (1981). "Algebraically specified programming systems and Hoare's logic". *Lecture Notes in Computer Science*, Vol. 115, 348-362.

- Chandy, K. & Misra J. (1988). *Parallel program design: a foundation*. Addison-Wesley.
- Clarke, E. (1985). "The characterization problem for Hoare logics". *Mathematical Logic and Programming Languages, C. A. R. Hoare y J. C. Shepherdson eds*., Englewood Cliffs, 89-106.
- Clint, M. & Hoare, C. (1972). "Program proving: jumps and functions". *Acta Informatica*, 1, 214-244.
- Constable, R. & O'Donnel, M. (1978). *A Programming Logic*. Wintrop.
- Cook, S. (1978). "Soundness and completeness of an axiom system for program verification". *SIAM J. Computing*, 7, 70-90.
- de Bakker, J. (1980). *Mathematical theory of program correctness*. Englewood Cliffs.
- Dijkstra, E. (1976). *A discipline of programming*. Prentice-Hall.
- Ehrig, H. & Mahr, B. (1985). "Fundamentals of algebraic specifications I: equations and initial semantics". *European Association for Theoretical Computer Science Monographs on Theoretical Computer Sciences*, Vol. 6, Berlin, Springer.
- Emerson, E. & Clarke, E. (1982). "Using branching time temporal logic to synthesize synchronization skeletons". *Sci. Comput. Programming*, 2, 241-266.
- Floyd, R. (1967). "Assigning meaning to programs". *Proc. American Mathematical Society, Symposium on Applied Mathematics*, 19, 19-32.
- Francez, N. (1983). "Product properties and their direct verification". *Acta Informatica*, 20, 329-344.
- Francez, N. (1986). *Fairness*. Springer.
- Francez, N. (1992). *Program verification*. Addison-Wesley.
- Gries, D. & Levin, G. (1980). "Assignment and procedure call proof rules". *ACM Trans. Prog. Lang. Syst*., 2(4), 564-579.
- Grumberg, O., Francez, N., Makowsky, J. & de Roever, W. (1985). "A proof rule for fair termination of guarded commands". *Information and Control*, 66, 83-102.
- Harel, D. (1979). "First Order Dynamic Logic". *Lecture Notes in Computer Science*, Vol. 36, Berlin, Springer.

- Hoare C. (1969). "An axiomatic basis for computer programming". *Comm. ACM*, 12, 576-580.
- Hoare, C. (1971a). "Proof of a program: FIND". *Comm. ACM*, 14(1), 39-45.
- Hoare, C. (1971b). "Procedures and parameters: an axiomatic approach". *Proc. Sym. on Semantics of algorithmic languages, E. Engeler ed., Lecture Notes in Mathematics*, Vol. 188, Berling, Springer.
- Hoare, C. & Wirth, N. (1973). "An axiomatic definition of the programming language PASCAL". *Acta Informatica*, 2, 335-355.
- Howard, J. (1976). "Proving monitors". *Comm. ACM*, 19(5), 273-279.
- Huth, M. & Ryan, M. (2004). *Logic in computer science*. Cambridge University Press.
- Jansen, T. (1983). "Foundations and applications of Montague grammar". *PhD Thesis*, University of Amsterdam.
- Jones, C. (1992). "The search for tractable ways of reasoning about programs". *Tech. Rep. UMCS-92-4-4*, Department of Computer Science, University of Manchester, Manchester.
- Loeckx, J. & Sieber, K. (1987). *The foundation of program verification*, second ed. Teubner-Wiley, Stuttgart.
- Manna, Z. & Pnueli, A. (1991). *The temporal logic of reactive and concurrent systems – specification*. Springer-Verlag.
- Manna, Z. & Pnueli, A. (1995). *Temporal verification of reactive systems – safety*. Springer-Verlag.
- Martin, A. (1983). "A general proof rule for procedures in predicate transformer semantics". *Acta Informatica*, 20, 301-313.
- Owicki, S. & Gries, D. (1976a). "An axiomatic proof technique for parallel programs". *Acta Informatica*, 6, 319-340.
- Owicki, S. & Gries, D. (1976b). "Verifying properties of parallel programs: an axiomatic approach". *Comm. ACM*, 19, 279-285.
- Plotkin, G. (1981). "A structural approach to operational semantics". *Technical Report DAIMI-FN*, 19, Computer Science Department, Aarhus University.
- Pnueli, A. (1977). "The temporal logic of programs". *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 46-57.

- Queille, J. & Sifakis, J. (1981). "Specification and verification of concurrent systems in CESAR". *Proceedings of the 5th International Symposium on Programming*, Paris.
- Rosenfeld, R. & Irazábal, J. (2010). *Teoría de la computación y verificación de programas*. EDULP, McGraw-Hill.
- Tucker, J. & Zucker, J. (1988). "Program correctness over data types with error-states semantics". *Centrum voor Wiskunde en Informatica Monograph 6*, Amsterdam, North-Holland.
- Woodcock, J., Gorm Larsen, P., Bicarregui, J. & Fitzgerald, J. (2009). "Formal methods: practice and experience". *ACM Computing Surveys*, 41, 4, 1-40.

Epílogo

Generalmente al comenzar y al terminar el dictado de la asignatura Teoría de la Computación y Verificación de Programas, les digo a mis alumnos que por aprender estos contenidos no les aseguro que vayan a conseguir trabajo enseguida como profesionales de la informática en el área de sistemas de un banco, por decir algo. La idea es provocar una discusión con ellos acerca de la importancia de estudiar fundamentos como los que aparecen en este libro (mal llamados a veces de manera discriminatoria "temas teóricos"). De esto se trata en última instancia el presente trabajo, de presentar y aplicar nociones tales como inducción, autómata, demostración por el absurdo, gramática, sistema axiomático, semántica de un lenguaje de programación, reducción de problemas, etc. A los alumnos llego a decirles que la asignatura es una excusa para enseñar en un determinado orden conceptos de esta naturaleza. Porque tarde o temprano, aún sin saberlo, emplearán estos conocimientos, para lograr la abstracción necesaria para resolver problemas computacionales, combinando componentes para obtener soluciones complejas a partir de soluciones más simples. La discusión obviamente se simplifica con los estudiantes con vocación para la investigación. De esta manera, *Computabilidad, Complejidad y Verificación de Programas* es un viaje de vuelta no sólo en el sentido del recorrido a través del universo de los problemas hacia su interior, sino también en el de aprovechar la madurez de los estudiantes con respecto a la algorítmica y las matemáticas para llevar a cabo un análisis más abstracto y profundo. Por fortuna, la profusión de material bibliográfico, potenciado por los buscadores en Internet que permiten encontrar cualquier elemento acá y ahora, facilitan sobremanera la tarea docente. Coincidentemente, en 2012 se cumplieron cien años del nacimiento de A. Turing, uno de los protagonistas principales de este libro. Un evento a mitad de año a propósito de dicho acontecimiento reunió a un gran número de investigadores, todos ellos receptores del Premio Turing (especie de Nobel de la computación). Traigo a colación este hecho porque un *leitmotiv* del encuentro fue revisar todo lo logrado desde el famoso artículo de Turing de 1936, al tiempo de plantear lo mucho que queda por hacer aún en temas básicos como la producción de software industrial confiable. Más que suficiente (si lo dicen ellos…) para motivar a estudiantes y docentes a la lectura y dictado de los tópicos de este trabajo.

Índice de definiciones

Índice de teoremas

Índice de ejemplos

Índice de ejercicios