Teoría de la computabilidad. Recursión, potencialidad y límites de las máquinas

Damiano Zanardini

Capítulo 1

Martes 28/09/10 (1 hora)

1.1 Quién soy yo

1.2 De qué va la asignatura

Nota: la palabra "recursión" es importante en este curso, pero no tanto como para que aparezca en el título. Podríamos decir que fue un despiste.

Lo que se puede calcular; ¿en qué lenguaje? ¿con qué procesador? ¿en qué sistema oprativo? En cualquiera. De hecho, vamos a ver varios formalismos para la computación y veremos que hacen todos lo mismo.

Necesitamos una definición de "calcular", y no va a ser demasiado fácil (Gödel tardó mucho en quedarse satisfecho).

1.3 Plan de trabajo; temario

- Diversos formalismos para expresar la idea de Computación
- El camino hacia la Tesis de Church
- Aritmetización de conceptos de la Teoría de Computabilidad
- Decidibilidad y Recursividad
- Aritmética. Los teoremas de Gödel
- Complejidad algorítmica (pocas cosas)
- Demostración Automática y Descubrimiento Científico
- Teoría de Computabilidad y Problema Mente-Máquina: comienzo de discusión
  - los puntos de vista de la inteligencia artificial

  - las tesis de Lucas y Penrose
  - escenarios fascinantes

- Teoría de Computabilidad y Análisis de Programas: problemas de cada día
  - análisis de terminación, análisis estático en general
  - uso de un analizador

1.4 Complejidad vs. computabilidad

1.4.1 Computabilidad

- problemas (in)solúbiles (sin límites de espacio-tiempo)
- propiedad de los problemas solúbiles
- propiedad de los problemas insolúbiles

1.4.2 Complejidad

- problemas solúbiles dado un límite de espacio y de tiempo
- dado un problema, ¿qué recursos necesito? (y si no los tengo, ¿existen soluciones aproximadas?)
- propiedad de problemas que están dentro de un límite
- ¿existen problemas más difíciles?
- ¿existen una solución optimal para un problema?

1.5 Bibliografía

- "Elements of the Theory of Computation" [10];
- "Computabilility" [5];
- "Computability Theory" [4];
- "Explaining the Ineffable" [16];
- "Theory of Recursive Functions and Effective Computability" [15]
- "Computability and Logic" [2]

1.5.1 Lecturas recomendadas

- "Sobre proposiciones formalmente indecidibles de los Principia Mathematica y sistemas afines" [7];
- "Le théorème de Gödel" [11];
- "The Undecidable" [6];
- "Classical Recursion Theory" [13];
- "Gödel, Escher, Bach" [8];
- "On Computable Numbers, with an Application to the Entscheidungsproblem" [17];
- "A Note on the Entscheidungsproblem" [3];
- "Computing Machinery and Intelligence" [18];
- "Estudio de los sistemas de descubrimiento científico automatizado" [1];
- "The Emperor's New Mind" [14];
- "Introducción a la Teoría de Autómatas, Lenguajes y Computación" [9];
- "Principles of Program Analysis" [12].

1.6 Grandes nombres (en orden más cronológico que de importancia)

- Georg Cantor (Cardinalidad de conjuntos infinitos, diagonalización);
- Jacques Herbrand (funciones recursivas);
- Kurt Gödel (funciones recursivas, incompletitud, aritmetización);
- Alonzo Church (λ-cálculo, tesis de Church-Turing);
- Alan Turing (máquinas, tesis de Church-Turing, mucho más);
- Stephen Kleene (λ-cálculo, teorema de recursión, equivalencias entre formalismos, expresiones regulares)
- John Rosser (λ-cálculo, equivalencias entre formalismos);
- Emil Post (máquinas de Post, grados de indecidibilidad);
- Henry Gordon Rice (teorema).

Nota: mucho de lo que vamos a ver se hizo en los años 30.

1.7 Empezamos: definición de algoritmo

- tiene un número *finito* de instrucciones
- la instrucciones tienen un efecto *limitado*
- la computación se desarrolla en pasos *individuales* y *discretos* (no analógicos ni continuos)
- los pasos son *deterministas*: dependen de un número (finito) de pasos previos y de una cantidad finita de datos
- no hay límite al número de pasos ni a la cantidad de memoria necesaria para almacenar los datos (finitos) iniciales, intermedios y finales (en este caso, notamos que el número de pasos sería finito)

Hoy en día, hay muchos planteamientos que no satisfacen del todo esta definición

- algoritmos paralelos;
- algoritmos randomizados/no deterministas;
- programas que se auto-modifican;
- algoritmos distribuídos;
- algoritmos no discretos (¿?);
- quantum computing;
- algoritmos genéticos (¿?).

Capítulo 2

Jueves 30/09/10 (2 horas)

2.1 Historia

- algoritmo de Euclide
  - segmentos conmensurables
  - quitar el más pequeño del más grande
    * Sean AB y CD los dos números que no son primos uno al otro. Se necesita entonces encontrar la máxima medida común de AB y CD.
    * Si CD mide AB entonces es una medida común puesto que CD se mide a sí mismo. Y es manifiesto que también es la mayor medida pues nada mayor a CD puede medir a CD.
    * Pero si CD no mide a AB entonces algún número quedará de AB y CD, el menor siendo continuamente restado del mayor y que medirá al número que le precede. Porque una unidad no quedarÃ¡ pues si no es asÃ, AB y CD serán primos uno del otro [Prop. VII.1], lo cual es lo contrario de lo que se supuso.
    * Por tanto, algún número queda que medirá el número que le precede. Y sea CD midiendo BE dejando EA menor que sÃ-mismo y sea EA midiendo DF dejando FC menor que sí mismo y sea FC medida de AE.
    * Entonces, como FC mide AE y AE mide DF, FC será entonces medida de DF. Y también se mide a sí mismo. Por tanto también medirá todo CD. Y CD mide a BE. Entonces CF mide a BE y también mide a EA. Así mide a todo BA y también mide a CD. Esto es, CF mide tanto a AB y CD por lo que es una medida común de AB y CD.
    * Afirmo que también es la mayor medida común posible porque si no lo fuera, entonces un número mayor que CF mide a los números AB y CD, sea éste G. Dado que G mide a CD y CD

mide a BE, G también mide a BE. Además, mide a todo BA por lo que mide también al residuo AE. Y AE mide a DF por lo que G también mide a DF. Mide también a todo DC por lo que mide también al residuo CF, es decir el mayor mide al menor, lo cual es imposible.

  - Por tanto, ningún número mayor a CF puede medir a los números AB y CD.
  - Entonces CF es la mayor medida común de AB y CD, lo cual se quería demostrar.

- Leibnitz (Siglo XVII): soñaba con una máquina que diera un valor de verdad a fórmulas matemáticas

- Hilbert, Paris, 1900: los 23 problemas (especialmente el décimo)

  - encontrar un **algoritmo** para determinar si una ecuación diofántica polinomial con coeficientes enteros tiene una solución entera
  - ecuaciones diofánticas: ecuaciones polinomiales como $x^2 + y^2 = z^2$

- Hilbert, 1928: el Entscheidungsproblem

  - el reto en lógica simbólica de encontrar un algoritmo general que decidiera si una fórmula del cálculo de primer orden es un teorema
  - en general, el problema se plantea para un lenguaje formal y una proposición matemática en este lenguaje
  - no importa si el algoritmo proporciona o no una solución, basta con que no falle nunca

- Enderton: "Princeton in the 1930's was an exciting place for logic. There was Church together with his students Rosser and Kleene. There was John von Neumann. Alan Turing who had been thinking about the notion of effective calculability, came as a visiting graduate student in 1936 and stayed to complete his Ph. D. under Church. And Kurt Gödel visited the Institute for advanced Study in 1933 and 1935, before moving there permanently."

- trabajo independiente de Church y Turing

  - Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem" [17].
  - Church. "A Note on the Entscheidungsproblem" [3].

- pero antes, pasó por allí Gödel

2.2 Máquinas de Turing

Nuestra referencia: "Elements of the Theory of Computation" [10].

- "On Computable Numbers, with an Application to the Entscheidungsproblem". 1936 (1937).
- en este artículo (unas 32 páginas con apéndice) se introduce las Máquinas de Turing, la máquina universal, los números computables, el problema de la parada y se aplica todo al Entscheidungsproblem
- se apoya en la aritmetización de Gödel

$$\langle K, \Sigma, s, H \subseteq K, \delta \rangle$$

- Σ son los símbolos (incluyendo el símbolo $\triangleright$ de inicio de la cinta; suponemos que toda máquina al encontrar este símbolo, se mueva inmediatamente hacia la derecha; y el símbolo $\llcorner$ del vacío);
- K son los estados;
- s es el estado inicial;
- H son los estados finales;
- δ es la función de computación. $\delta : (K \setminus H) \times \Sigma \mapsto K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$

Hay *configuraciones* (descripciones instantáneas) y *acciones* ($\leftarrow$ es ir-a-izquierda, $\rightarrow$ es ir-a-derecha). Para empezar, sólo vamos a tener el símbolo 1.
Ejemplos de máquinas:

- función constante (escribe un número fijo)
- suma de dos números
- producto
- máquina que no termina (siempre se mueve hacia la derecha)

Discusión

- hay que especificar un formato para el input (¿qué pasaría si hubiera símbolos no vacíos en algún punto de la cinta?)
- ¿cuántos símbolos se necesitan?
- ¿es igual no poder dar ningún paso más o parar en un estado final?

Extensiones (no añade potencia computacional)

- cinta infinita en los dos lados
- más de una cinta
- no determinista

Capítulo 3

Martes 05/10/10 (1 hora)

3.1 Máquinas de Turing (2)

- hemos visto ejemplos sencillos
- y uno complejo
- hemos visto que hay que elegir un formato para el input y para el output
- hemos intuido que
  - definir computaciones es mucho trabajo
  - se necesita una notación más ligera y que permita definir máquinas de forma *composicional*

Notas

- δ es una función con dominio $(K \setminus H) \times \Sigma$; es decir, tiene que estar definida para todos los estados que no sean finales y todos los símbolos
- si $\delta(q_1, \triangleright) = (q_2, b)$ entonces b tiene que ser $\rightarrow$ (está en la definición)
- si $\delta(q_1, a) = (q_2, b)$ entonces b no puede ser $\triangleright$ (está en la definición)

Configuraciones Una *configuración* de una máquina $M = (K, \Sigma, \delta, s, H)$ es un elemento de $K \times \triangleright\Sigma^* \times (\Sigma^*(\Sigma \setminus \{\sqcup\}) \cup \{e\})$ donde e es la secuencia vacía.

Para entender esta definición hay que conocer la estrella de Kleene y la notación para concatenar secuencias de símbolos.

Si el estado es final, entonces la configuración también es *final*.

Notación compacta: $(q, \triangleright ab\underline{c}d)$ por $(q, \triangleright abc, d)$.

Pasos

$$(q_1, w_1\underline{a_1}u_1) \vdash (q_2, w_2\underline{a_2}u_2)$$

si y sólo si para un $b \in \Sigma \cup \{\leftarrow, \rightarrow\}$ y con $\delta(q_a, a_1) = (q_2, b)$:

- $b \in \Sigma$, $w_1 = w_2$, $u_1 = u_2$ y $a_2 = b$; o bien
- $b = \leftarrow$, $w_1 = w_2 a_2$, y una de las dos
  - $u_2 = a_1 u_1$ si $a_1 \neq \sqcup$ o bien $u_1 \neq e$;
  - $u_2 = e$ si $a_1 = \sqcup$ y $u_1 = e$;
- $b = \rightarrow$, $w_2 = w_1 a_1$, y una de las dos
  - $u_1 = a_2 u_2$;
  - $u_1 = u_2 = e$ y $a_2 = \sqcup$.

El input Suponemos que la configuración de input es $(s, \triangleright\underline{\sqcup}w)$ donde w es el input (puede ser vacío).

Ejemplos

- la máquina que no termina
- la máquina que copia (multiplica por 2)

Capítulo 4

Jueves 07/10/10 (2 horas)

4.1 Composición de Máquinas de Turing

Ejemplos

- la máquina que no termina

Componer máquinas

- hemos visto que hay operaciones que se repiten muy a menudo:
  - escribir un símbolo;
  - moverse a la derecha o a la izquierda;
  - moverse a un lado hasta encontrar un símbolo σ
  - moverse a un lado hasta encontrar un símbolo que no sea σ
- y definimos máquinas sencillas que sólo hacen una operación y paran:
  - escribir un símbolo: M_a o sencillamente a;
  - moverse a la derecha o a la izquierda: $M_\rightarrow$ o R, $M_\leftarrow$ o L;
- y una notación gráfica para combinar máquinas

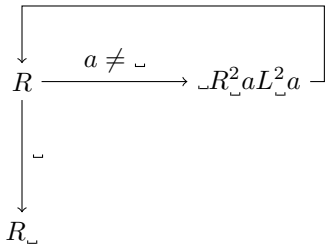
$$\begin{array}{l} M_1 \xrightarrow{\ a\ } M_2 \\ \Big\downarrow b \\ M_3 \end{array}$$

  - ejecuta M_1 hasta que termine la computación
  - si el símbolo leído en la configuración final de M_1 es a, empieza M_2 desde el mismo sitio

  - si el símbolo leído en la configuración final de M_1 es b, empieza M_3 desde el mismo sitio

- los conjuntos de estados de cada máquina son disjuntos
- (definición formal de la composición)
- una flecha con más de un símbolo es una abreviación de varias flechas
- si una flecha tiene todos los símbolos, entonces no se escriben
- otras máquinas "derivadas":
  - $R_\sqcup$ y $L_\sqcup$ (hasta que encuentra un $\sqcup$)
  - $R_{\overline{\sqcup}}$ y $L_{\overline{\sqcup}}$ (hasta que encuentra algo que no sea $\sqcup$)
  - en general, R_a, L_a, $R_{\overline{a}}$, $L_{\overline{a}}$

Ahora podemos escribir la máquina que copia que vimos la otra vez: de $\underline{\sqcup}w\sqcup$ a $\sqcup w\sqcup w\underline{\sqcup}$.



Notamos que en la δ de la máquina global hay que "acordarse" del símbolo a que se ha visto.

Aceptar lenguajes

- un format estándar de input
- dos estados finales y y n
- puede acabar en una *configuración de aceptación* o de *rechazo*
- M *acepta* o *rechaza* un input w
- se dice que una máquina M *decide* un lenguaje $L \subseteq \Sigma_0^*$ (donde $\Sigma_0 = \Sigma \setminus \{\sqcup, \triangleright\}$) si

Capítulo 5

Jueves 14/10/10 (2 horas)

5.1 Primeros elementos de computabilidad

Las tareas de una MdT y de todos los formalismos de computación

- aceptar lenguajes
  - un formato para el input
  - dos estados finales: y, n
- calcular funciones
  - un formato para el input y el output

Recursividad

- decidir un lenguaje; lenguajes recursivos
- semidecidir un lenguaje; lenguajes recursivamente enumerables
- recursivamente enumerable implica recursivo
- el complemento de un recursivo es recursivo
- el complemento de un recursivamente enumerable no es recursivamente enumerable

5.2 Extensiones de las Máquinas de Turing

MdT con k cintas

- características
- copia de una palabra: cómo trabaja una MdT con 2 cintas
- simulación de las k cintas por medio de una cinta: demostración constructiva de que el poder de calcular funciones o aceptar lenguajes no cambia

Cinta infinita en los dos lados

- discusión informal

No determinismo

- comienzo de discusión; a continuar...

Capítulo 6

Jueves 21/10/10 (2 horas)

6.1 Máquinas de Turing no deterministas

MdT no deterministas Necesitamos mucho cuidado con la definiciones de qué significa aceptar un lenguaje y calcular una función.

Una MdT no determinista se define como una normal, pero Δ (que sustituye a δ) es una *relación*:

$$\Delta \subseteq ((K \setminus H) \times Sigma) \times (K \times (\Sigma \cup \{\leftarrow, \rightarrow\}))$$

Una configuración puede generar más de una en un único paso.

Aceptar un input

- una MdT acapta un input w si existe al menos un camino de la configuración inicial hasta una con el estado y (no importa si también las hay que terminano en n)
- $(s, \triangleright \underline{\sqcup} w) \vdash^* (y, \triangleright u \underline{a} v)$

Semidecidir un lenguaje

- M semidecide un lenguaje L si y sólo si: para todo input w, $w \in L$ sii M acepta w

Decidir un lenguaje

- M decide un lenguaje L si y sólo si: para todo input w
  - existe un número natural N (que depende de M y w) tal que no hay ninguna configuración C con $(s, \triangleright \underline{\sqcup} w) \vdash^N C$ (es decir, todas las computaciones terminan)
  - $w \in L$ si y sólo si $(s, \triangleright \underline{\sqcup} w) \vdash^* (y, \triangleright u \underline{a} v)$ para ciertos u, v, a

  es decir, basta que una computación termine en y

Calcular una función

- M calcula una función f si y sólo si: para todo input w
  - existe un número natural N (que depende de M y w) tal que no hay ninguna configuración C con $(s, \triangleright\underline{\sqcup}w) \vdash^N C$ (es decir, todas las computaciones terminan)
  - $(s, \triangleright\underline{\sqcup}w) \vdash^* (h, \triangleright u\underline{a}v)$ si y sólo si $u = \sqcup$ y $v = f(w)$ (es decir, se requiere que *todas* las computaciones terminen con el output correcto)

En la aceptación de lenguajes, las respuestas positivas prevalecen sobre las negativas (es una definición asimétrica): basta que haya una respuesta positiva para aceptar.

Esto implica que no se pueda hacer el truco del intercambio y - n para (semi)decidir el lenguaje complemento.

En cambio, para calcular funciones se requiere que todas las computaciones den el mismo resultado, porque sería imposible decidir cuál de las respuestas es la correcta.

Queremos demostrar que existe una MdT determinista que actúa igual que una no determinista.

El poder del no determinismo La MdT que dice si un número N es compuesto (es decir, no es primo y no es tampoco 0 ni 1). En lugar de intentar secuencialmente con todos los número hasta $\sqrt{N}$, se puede definir una MdT no determinista que semidecide el lenguaje C de los números compuestos (escritos en notación binaria) *intentándolo* con todos los factores posibles.

- elegir no determinísticamente dos números y escribirlos en notación binaria al lado del input
- multiplicar los dos números
- ver si el input y el producto son el mismo número; si no lo son, no terminar (estamos hablando de semidecidir C)

Se puede modificar esta MdT para que *decida* C:

- en el primer paso, nunca elige dos números con más bits que N
- en el tercero, en lugar de no terminar, termina en el estado n (es decir, se puede demostrar que todas las computaciones terminan)

Demostrar la equivalencia Si una MdT no determinista M semidecide o decide un lenguaje o calcula una función, entonces existe una MdT determinista M' que hace lo mismo.

(vamos a ver el caso de semidecidir)

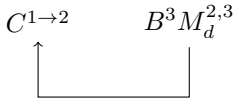
Dado un input w, M' intenta ejecutar sistemáticamente todas las computaciones de M a través de un proceso de *dovetailing*. El punto es que, dada una

configuración C, el número de configuraciones que pueden seguir en M es fijo (limitado por $r = |K| \cdot (|\Sigma| + 2)$).

Tenemos M_d que tiene dos cintas: la primera es la de M, en la segunda hay n números de 1 a r: $i_1 i_2 .. i_n$. Simulando el paso j de M, M_d lee el j-ésimo número i_j de la segunda cinta y ejecuta la alternativa número i_j de M (estará escrita en su δ). Luego va al número siguiente y continúa con el proceso. Si termina de leer la segunda cinta para.

M' tiene tres cintas:

- la primera nunca cambia y contiene el input
- la segunda y la tercera se usan para simular M_d para toda secuencia $i_1 i_2 .. i_n$ de $\{1..r\}^*$
  - el input es copiado de la primera cinta a la segunda antes de que M' empiece a simular cada computación
  - inicialmente la tercera cinta es vacía (primer elemento de $\{1..r\}^*$)
- entre dos simulaciones de M_d, una MdT auxiliar genera las secuencias de $\{1..r\}^*$ léxicograficamente



- La MdT $C^{1\to 2}$ es la que copia el input de la primnera a la segunda cinta
- B^3 genera los elementos de $\{1..r\}^*$ en la tercera cinta
- $M_d^{2,3}$ es M_d que trabaja en la segunda y en la tercera cinta

Se puede demostrar que M' para con el input w sii alguna compuatción de M lo hace

- el único modo en que M para es que $M_d^{2,3}$ NO pare por haber encontrado un $\llcorner$ (si no, M continuaría con $C^{1\to 2}$); esto significa que hay una computación de M que termina
- la otra dirección también vale: si una computación de M termina, entonces tendrá como mucho n pasos, y M_d terminará (no encontrando un $\llcorner$) con un elemento de $\{1..r\}^n$ en la tercera cinta

6.2 Funciones (numéricas) recursivas

Ahora no nos vamos a preocupar de *cómo* se calcula, sino solamente de lo que se calcula: funciones de números a números.

$$f(m, n) = m \cdot n^2 + 3 \cdot m^{2m+17}$$

¿Cómo sabemos que esta función se puede calcular? Porque es la composición de sumas, productos y exponenciales. Y el exponencial se define *recursivamente* con el producto y la misma función aplicada a números más pequeños.

Vamos a empezar con funciones sencillas (por las que está claro que se pueden calcular) y luego las compondremos para obtener funciones más complejas. Veremos lo que podemos calcular con estas herramientas.

- funciones básicas:
  - $zero_k(n_1, .., n_k) = 0$ para todo $n_1, .., n_k$
  - $id_{k,j}(n_1, .., n_k) = n_j$
  - $succ(n) = n + 1$
- formas de combinarlas
  - composicioón: dadas g (con aridad k) y $h_1..h_k$ (con aridad l) se puede definir su composición como la función

$$f(n_1, .., n_l) = g(h_1(n_1, .., n_l), .., h_k(n_1, .., n_l))$$

  - dada g (con aridad k) y h (con aridad $k + 2$), la función definida *recursivamente* con g y h es la f (con aridad $k + 1$)

$$\begin{aligned} f(n_1, .., n_k, 0) &= g(n_1, .., n_k) \\ f(n_1, .., n_k, m + 1) &= h(n_1, .., n_k, m, f(n_1, .., n_k, m)) \end{aligned}$$

Con estas reglas se definen las funciones *primitivas recursivas*.

Ejemplos

- $plus_2(n) = n + 2$
- $plus(m, n) = m + n$
- $mult(m, n) = m \cdot n$
- $exp(m, n) = m^n$
- funciones constantes
- $sign(n)$

Capítulo 7

Martes 26/10/10 (1 hora)

7.1 Funciones primitivas recursivas

- predicado *isZero*

$$\begin{aligned} isZero\,(0) &= succ\,(zero_1\,(0)) = 1 \\ isZero\,(m+1) &= zero_3\,(n, n, isZero\,(m)) \end{aligned}$$

- resto "-1"

$$\begin{aligned} sub_1\,(0) &= zero_1\,(0) \\ sub_1\,(m+1) &= ID_{3,2}\,(n, m, sub_1\,(m)) = m \end{aligned}$$

- resto

$$\begin{aligned} sub\,(n, 0) &= ID_{2,1}\,(n, 0) = n \\ sub\,(n, m+1) &= sub_1\,(ID_{3,3}\,(n, m, sub\,(n, m))) \end{aligned}$$

- predicado "igual a 3"

$$\begin{aligned} eq^3\,(n) = isZero(sum(&sub\,(n, succ\,(succ\,(succ\,(zero_1\,(n)))))\,, \\ &sub\,(succ\,(succ\,(succ\,(zero_1\,(n))))\,, n))) \end{aligned}$$

- predicado *greaterThanOrEqual*

$$greaterThanOrEqual\,(n, m) = isZero\,(sub\,(m, n))$$

- función definida por casos, usando un predicado

$$f(n_1, .., n_k) = \begin{cases} g(n_1, .., n_k) & \text{si } p(n_a, .., n_k) \\ h(n_1, .., n_k) & \text{si no } p(n_a, .., n_k) \end{cases}$$

  se puede definir como

  $$f(n_1, .., n_k) = p(n_1, .., n_k) \cdot g(n_1, .., n_k) + (1 - p(n_1, .., n_k)) \cdot h(n_1, .., n_k)$$

  (donde "$-$" sería el resto no-negativo)

Capítulo 8

Jueves 28/10/10 (2 horas)

8.1 El lenguaje *for*

Definimos el *lenguaje for*, imperativo, que básicamente incluye asignación, skip, concatenación y bucles, y trabaja con enteros.

$$\begin{array}{lcl} E & ::= & n|X|E+E|E\times E|E-E|F \\ B & ::= & true|false|E<E|\neg B|B\vee B \\ C & ::= & \mathsf{skip}|X:=E|C;C|\{C\}\ \textbf{if}\ B\ \textbf{then}\ C\ \textbf{else}\ C \\ & | & \textbf{for}\ X=E\ \mathsf{to}\ E\ \textbf{do}\ C|\textbf{return}\ E \\ F & ::= & f(X,..,X)\{C\} \\ X & ::= & x_1|x_2|... \end{array}$$

Algunas notas:

- un programa es un conjunto de declaraciones de funciones: procedimientos que devuelven un valor;
- como en Pascal, las expresiones que determina n el número de iteraciones del **for** se evalán antes del bucle y sólo una vez; por lo tanto, el número de iteraciones se sabe antes de empezar;
- no se admite recursión;
- cosas com la igualdad o la conjunción se definen fácilmente a partir de los operadores dados.

Con estas condiciones, y con la semántica usual, cualquier programa escrito en el lenguaje *for* termina.

Para nuestros objetivos, definimos también el lenguaje for_r, igual a *for* pero que admite recursión. Está claro que hay programas escritos en for_r que no terminan.

Relación con las primitivas recursivas Podemos implementar fácilmente las funciones primitivas recursivas en el lenguaje for_r:

- en general:

```
1  f(n1,..,nk,m) {
2    if (m=0) then return g(n1,..,nk)
3    else return h(n,m,f(n1,..,nk,m-1))
4  }
```

- ejemplo: *sub*

```
1  sub(n,m) {
2    if (m=0) then return n
3    else return sub1(sub(n,m-1))
4  }
```

Eliminar la recursión Eliminando la recursión, estas funciones se implementan en *for*. Notamos que la eliminación de la recursión es un resultado que se puede conseguir siempre con método sistemáticos.

- en general:

```
1  f(n1,..,nk,m) {
2    returnvalue := g(n1,..,nk);
3    for i=0 to m-1 do {
4      returnvalue := h(n1,..,nk,i,returnvalue);
5    };
6    return returnvalue;
7  }
```

- ejemplo: *sub*

```
1  sub(n,m) {
2    returnvalue := n;
3    for i=0 to m-1 {
4      returnvalue := sub1(returnvalue);
5    };
6    return returnvalue;
7  }
```

Esto nos dice que *for* describe al menos todas las funciones primitivas recursivas.

8.2 Límites de las primitivas recursivas

Usando los ingredientes de las funciones primitivas recursivas (funciones básicas, composición, recursión) se puede definir muchas funciones interesantes. Pero, por medio de la *diagonalización*, se demuestra que las funciones primitivas recursivas no pueden con todo lo *calculable*.

- nos limitamos a funciones con un argumento; el resultado es generalizable a funciones con más argumentos;
- las funciones primitivas recursivas son enumerables, es decir, a cada una se le puede asignar un número natural sin repeticiones; son numerables porque
  - la definición de una función es una secuencia finita de símbolos, por lo que se puede enumerar estas secuencias según un orden;
  - de esta secuencia es fácil quitar (por medio de un control *sintáctico*) las descripciones que no describes funciones primitivas recursivas (p.ej. "(()()—())" es una de ellas);
- podemos escribir la n-ésima función f_n como la lista (infinita) de todos los valores que devuelve:

$$f_n(0), f_n(1), .., f_n(17), ..$$

- si representamos así todas las funciones obtenemos una tabla con un número infinito (enumerable) de filas infinitas (enumerables):

$$\begin{array}{ccccccc} f_0(0) & f_0(1) & f_0(2) & ... & f_0(7) & ... \\ f_1(0) & f_1(1) & f_1(2) & ... & f_1(7) & ... \\ f_2(0) & f_2(1) & f_2(2) & ... & f_2(7) & ... \\ ... & ... & ... & ... & ... & \\ f_{14}(0) & f_{14}(1) & f_{14}(2) & ... & f_{14}(7) & ... \\ ... & ... & ... & ... & ... & \end{array}$$

- ahora consideramos una función f tal que, para todo n, $f(n) = f_n(n) + 1$; es decir, f representa la *diagonal* de la tabla;
- f es *total*;
- supongamos que f sea una de las funciones recursivas que la tabla representa, pongamos que sea f_m por un cierto m; entonces $f_m(m) = f(m) = f_m(m) + 1$ que es imposible;
- hemos derivado una contradicción, por lo que f no es una función primitiva recursiva;
- sin embargo, f es calculable: para un input n, basta con calcular $f_n(n)$ y sumar 1;
- entonces se concluye que **hay funciones calculables y totales que no son primitivas recursivas**

El típico ejemplo es la función de Ackermann. En general, un formalismo que sólo expresa funciones totales no las puede expresar todas.

La diagonalización

- es una herramienta matemática para hacer demostraciones que fue usada por Georg Cantor (otro gran nombre de esta asignatura) en 1891 para demostrar que hay conjuntos de números por los que los números reales no son contables
- después de aplicarla a las funciones primitivas recursivas, podríamos pensar que se puede aplicar a *cualquier* formalismo de computación, por lo que siempre podemos encontrar funciones calculables que no se calculan con dicho formalismo
- sin embargo no es así: ¿dónde está el truco?
- el truco está en que la diagonalización funciona para lor formalismos que sólo definen funciones *totales*, es decir, que siempre terminan...

8.3 Funciones μ-recursivas

Para solucionar este límite de las funciones primitivas recursivas introducimos una nueva forma de definir funciones:

$$f(n_1,..,n_k) = \begin{cases} \text{el mínimo número } m \text{ tal que } g(n_1,..,n_k,m) = 1 & \text{si existe} \\ 0 & \text{si no existe} \end{cases}$$

Una cosa importante e intuitiva es que esta forma de iteración corresponde al bucle **while**. La *minimización* de g se denota

$$\mu m[g(n_1,..,n_k,m) = 1]$$

No existe un método para calcular esta minimización, aunque sepamos cómo calcular g para todo input: si m no existe, entonces la única forma de saberlo sería intentar con todos los números naturales. Este corresponde al programa

```
1 m := 0;
2 while (¬(g(n1,..,nk,m) = 1)) do m := m+1;
3 return m;
```

Una función (predicado) g es *minimizable* si este algoritmo termina para todo $n_1,..,n_k$, es decir, si

$$\forall n_1,..,\forall n_k \exists m \; g(n_1,..,n_k,m) = 1$$

Una función se dice μ*-recursiva* si se define a partir de las funciones básicas, de la composición, la recursión y la minimización de funciones *minimizables*.

Esta última palabra es la más importante en la definición.

Ejemplo de función μ-recursiva A través de la minimización podemos definir el logaritmo $\log_{m+2}(n+1)$:

$$log\,(m,n) = \mu p[mayor - igual\,((m+2)^p, n+1)$$

Notamos que el logaritmo se puede también calcular con una primitiva recursiva (es decir, sin minimización).

Tenemos un resultado importante a demostrar sobre las funciones μ-recursivas, pero antes vamos a hablar un poco de los lenguajes de programación que hemos empezado a usar.

El razonamiento diagonal Se podría pensar que, por medio de la diagonalización, vamos a poder encontrar una función calculable que no sea μ-recursiva. Al fin y al cabo

- una función μ-recursiva es total porque se define a través de la minimización de funciones **minimizables**: nunca devuelve un 0 por no haber encontrado el m;
- estas funciones siguen siendo definidas por cadenas finitas de símbolos;
- entonces, enumerando estas funciones, se podría aplicar la diagonalización.

Pero esto no funciona:

- en el caso de las primitivas recursivas, la enumeración se hacía considerando cada definición d_n de f_n como una secuencia finita de símbolos; de hecho, se podía decidir *sintácticamente* si una secuencia de símbolos d es una definición correcta de alguna f_n;
- sin embargo, aquí **no** se puede decidir sintácticamente si d define una función μ-recursiva, porque no se sabe, en general, si la g que aparece en d es minimizable.

En general, la diagonalización funciona para formalismos que *sólo* pueden expresar funciones *totales*. A pesar de que las μ-recursivas sean funciones totales, el formalismo que las expresa también expresa funciones parciales (cuando g no es minimizable), y no se puede en general decidir algorítmicamente si una funciíon es total.

Entonces

- si intentamos aplicar la diagonalización a la tabla de las μ-recursivas, no podemos decidir algorítmicamente si una función está en la lista; por lo tanto, la f que se construye en la diagonal no tiene por qué ser calculable;
- si intentamos aplicar la diagonalización a la tabla de *todas* las funciones que el formalismo expresa, entonces tendremos funciones parciales en la lista; la función f viene a ser

$$f(n) = \begin{cases} f_x(x)+1 & \text{si } f_x(x) \text{ está definida} \\ \text{indefinida} & \text{si } f_x(x) \text{ no está definida} \end{cases}$$

Supongamos entonces que f sea representada por el m-ésimo algoritmo: entonces, no podemos concluir la contradicción $f_m(m) = f(m) = f_m(m)+1$ (es decir, que $f \neq f_m$) porque f_m puede ser parcial.

Y ¿si extendieramos todas las funciones parciales con ceros para que se conviertan en funciones totales? No se puede hacer porque no siempre hay un algoritmo que calcula la función extendida.

8.4 El lenguaje *while*

El lenguaje *while* es como el *for* pero con

$$\begin{array}{rcl} C & ::= & \mathsf{skip} \mid X := E \mid C; C \mid \{C\} \textbf{ if } B \textbf{ then } C \textbf{ else } C \\ & \mid & \textbf{for } X = E \text{ to } E \textbf{ do } C \mid \textbf{return } E \mid \textbf{while } B \textbf{ do } C \end{array}$$

Informalmente hemos visto ya que

- las funciones (parciales) que se obtienen añadiendo a las primitivas recursivas la minimización son equivalentes a este lenguaje; y
- las μ-recursivas (que son totales) son equivalentes a las funciones totales implementables en *while*.

Nuestra intuición nos dicen que los lenguaje de programación reales son mucho más complejos, y nos preguntamos si las funciones calculadas por ellos siguen siendo las mismas. Los lenguaje reales incluyen:

- tipos, varios formatos de datos
- objetos
- varias estructuras para gestionar el flujo de control
- concurrencia
- recursion
- input-output

Vamos a verlo por puntos.

- estamos acostumbrados a pensar que todo el contenido de un disco duro es una secuencia de bits, por lo que no nos extraña pensar que, al fin y al cabo, todo tipo de dato se puede representar con números enteros naturales;
- todo lo que es la programación a objetos, lo sabemos, aumenta más que nada la *comodidad*, no las potencialidades
- idem por los diferentes constructos de iteración;
- hemos visto hace poco que la recursión se puede eliminar;
- interactividad en el siguiente párrafo.

Programas interactivos Quizá lo más difícil es aceptar que un programa interactivo no calcula nada más (desde nuestro punto de vista) que uno no interactivo. Una forma de convencerse de ello es pensar en cómo se proporciona el input al programa durante la ejecución:

- si el input que el usuario proporciona durante la computación no depende en ninguna forma de los resultados parciales obtenidos (es decir, si estaba todo preparado desde el principio), entonces se puede pensar que todo el input es dado antes de empezar la ejecución;
- en cambio, si el usuario introduce datos de una forma que depende del output parcial, entonces podemos pensar que esta dependencia es "una parte" del programa; es decir, si incorporamos en el programa la parte del usuario que genera nuevos input, tendríamos un programa no interactivo que hace lo mismo;
- si el input consiste de las dos cosas, se pueden combinar los enfoques.

Sin embargo, asumiendo que haya una dependencia entre el output parcial de una computación y los inputs siguientes, y que esta dependencia se pueda empotrar en el programa, estamos diciendo que la dependencia es calculable, lo que (tratándose de usuarios humanos) es el contenido de este curso y no es nada descontado.

En todo caso, suponiendo que la dependencia no sea calculable porque el hombre *puede más* que la máquina, nos tendríamos que preguntar si este *plus* que calculamos de esta forma se puede considerar como parte de la computación. Mi opinión es que no se puede, pero necesitaríamos reflexionar un poco más sobre el tema.

- *Interaction, Computability, and Church's Thesis.* Peter Wegner, Dina Goldin, 1999.

8.5 Nuestros tres formalismos, hasta ahora

- Máquinas de Turing: la base para descripción de la máquinas en general y para la complejidad; su ventaja es la **intuición**;
- funciones μ-recursivas: la base para programación funcional (con el λ-cálculo) y la semántica denotacional; su ventaja es la **claridad**;
- lenguaje *while*: la base para la programación imperativa y para la semántica operacional; su ventaja es la **familiaridad**;
- si nos dará tiempo, también hablaremos del λ-cálculo, la verdadera basa para la programación funcional; su ventaja es la **elegancia** y la **concisión**.

Capítulo 9

Martes 02/11/10 (1 hora)

9.1 Equivalencia entre Máquinas de Turing y funciones μ-recursivas

Una función total $f : N \mapsto N$ es μ-recursiva si y sólo si existe una MdT M_f que la calcula

Demostración del "sólo si" Supongamos que f sea una función μ-recursiva, es decir definida por medio de las funciones básicas y de la composición, la recursión y la minimización sobre predicados minimizables.

- Es fácil ver que las funciones básicas son Turing-calculables.
- supongamos que f tenga aridad k y sea la composición de g (aridad l) y $h_1, .., h_l$ (todas con aridad k), y que, por inducción, sepamos cómo calcular g y las h con una MdT. Entonces, la MdT que calcula f se define como la M con tres cintas sabemos que esto no cambia el resultado) que:
  - al principio tiene los k input en la primera cinta, y la segunda está vacía;
  - para $i \in \{1, .., l\}$, el input es copiado a la segunda cinta, y la máquina M_h^i que calcula h_i trabaja en la segunda cinta, terminando con la copia de su output a la tercera cinta (sin sobreescribir los resultados anteriores);
  - la tercera cinta contendrá el input para la MdT M_g que calcula g; esta máquina trabaja en la tercera cinta, y al final copia el resultdo a la primera.
- si f se define recursivamente de g y h, y M_g y M_h son las MdT correspondientes, entonces M_f tiene dos cintas; la primera contiene el input $n_1..n_k$ y se usa también para guardar los resultados temporales; la segunda se usa para calcular los resultados; M_f

  - calcula $g(n_1, .., n_k)$ y lo guarda en la primera cinta sin borrar el input; es esta área de la cinta se guardará el resultado parcial r de la computación;
  - escribe al final de la primera cinta el número 0, que se usará como contador i;
  - compara i con m: si son iguales, entonces devuelve r como output en la primera cinta (es decir, borra lo demás);
  - si el contador es menor, copia $n_1..n_k$ y r e i en la segunda cinta y ejecuta M_h; el resultado se vuelve a guardar en r, luego se incrementa i y se repite el proceso.

  Es fácil ver que la computación es simplemente lo que hace el lenguage *for* para simular la recursión.

- Si f se obtiene minimizando un predicado g (calculado por M_g), entonces M_f llama a M_g sobre el input (donde el útlimo argumento se incrementa a partir de 0), hasta que el valor del contador es devuelto como output la primera vez que M_g devuelve 1.

Notamos que, básicamente, esta demostración dice (muy informalmente) que una MdT hace todo lo que hace el lenguaje *while*.

Demostración del "si" Supongamos que $M_f = (K, \Sigma, \delta, s, \{h\})$ calcule una función (*unaria*, para que sea más simple; el caso de más argumentos es fácil) f. Asumimos que K y Σ sean disjuntos. Sea $b = |\Sigma| + |K|$, y sea E una función de $\Sigma \cup K$ a $\{0, .., b-1\}$ tal que $E(0) = 0$ y $E(1) = 1$ (como hablamos de funciones numéricas, 0 y 1 estarán en Σ).

Con estos ingredientes queremos representar configuraciónes con números en base b: la configuración $(q, \triangleright a_1 a_2 .. \underline{a_k} .. a_n)$ será representada por el número $a_1 a_2 .. a_k q a_{k+1} .. a_n$, es decir, por

$$E(a_1)b^n + E(a_2)b^{n-1} + .. + E(a_k)b^{n-k+1} + E(q)b^{n-k} + E(a_{k+1})^{b-k-1} + .. + E(a_n)$$

Ahora, f será definida como

$$f(n) = num\,(output\,(last\,(comp\,(n))))$$

donde

- $comp\,(n)$ es el número cuya representación en base b es la concatenación de la (única) secuencia de configuraciones que empieza por $(s, \triangleright_{\sqcup} w)$, donde w es n en base 2, y termina con $(h, \triangleright_{\sqcup} w')$, donde w es $f(n)$ en base 2; esta secuencia existe porque hemos supuesto que M_f calcula f;

- *last* toma este número y devuelve la última configuración;

- *output* borra de una configuración final $(h, \triangleright \underline{\ } w')$ (un número en base b) lo que no es w';
- *num* convierte la representación binaria de un número (una secuencia de 0 y 1) en el valor del número.

Hay que definir estas funciones y algunas funciones auxiliarias ($\sim$ es el resto no-negativo; $\log(b \sim 2, n \sim 1)$ no es nada más que $\log_b(n)$ aproximado por arriba; y la constante b es una propiedad de M_f):

$$\begin{array}{rcl} \mathit{digit}\,(d,m) & = & \mathit{rem}\left(\mathit{div}\left(m, b^{d-1}\right), b\right) \\ \mathit{lastpos}\,(m) & = & \mu d[\mathit{eq}\,(\mathit{digit}\,(d,m), E(\triangleright)) \vee \mathit{eq}\,(d,m)] \\ \mathit{last}\,(m) & = & \mathit{rem}\left(m, b^{\mathit{lastpos}(m)}\right) \\ \mathit{output}\,(c) & = & \mathit{rem}\left(c, b^{\log(b\sim 2, c\sim 1)\sim 2}\right) \\ \mathit{num}\,(n) & = & 2 \times \mathit{digit}\,(1,n) + 4 \times \mathit{digit}\,(2,n) + ..+ \\ & & (2^{\log(b\sim 2, n\sim 1)}) \times (\mathit{digit}\,(\log(b \sim 2, n \sim 1), n)) \end{array}$$

- $\mathit{digit}\,(d,m)$ devuelve la cifra que está en la posición d (empezando por la derecha) del número m, considerado en base b (p.ej., si $b = 10$, entonces $\mathit{digit}\,(3, 679576) = 5$);
- $\mathit{lastpos}\,(m)$ devuelve la última (empezando por la izquierda) posición en la representación de la secuencia de configuraciones donde aparece $\underline{\ }$ (se añade el caso donde $\underline{\ }$ no aparece, para que la función sea total);
- $\mathit{last}\,(m)$ básicamente borra las cifras a la izquierda de $\mathit{lastpos}\,(m)$;
- $\mathit{output}\,(c)$ borra las primeras tres cifras de c, es decir, en nuestro caso, $\triangleright$, $\underline{\ }$ y h;
- notamos que ésta no es exactamente la definición de $\mathit{num}\,(n)$, que de todas formas es primitiva recursiva.

La función más difícil de definir es $\mathit{comp}\,(n)$, que calcula, dado n, la secuencia de configuraciones de M_f que lleva a $f(n)$.

$$\begin{array}{rcl} \mathit{getState}\,(c) & = & \mathit{digit}(\mu d[\mathit{eq}\,(\mathit{digit}\,(d,c), E(q_1)) \vee .. \\ & & \vee \mathit{eq}\,(\mathit{digit}\,(d,c), E(q_k))], c) \\ \mathit{getSymbol}\,(c) & = & \mathit{digit}(1 + \mu d[\mathit{eq}\,(\mathit{digit}\,(d,c), E(q_1)) \vee .. \\ & & \vee \mathit{eq}\,(\mathit{digit}\,(d,c), E(q_k))], c) \\ \mathit{step}\,(c) & = & \mathit{doStep}(c, \mathit{deltaState}\,(\mathit{getState}\,(c), \mathit{getSymbol}\,(c)), \\ & & \qquad \mathit{deltaSymbol}\,(\mathit{getState}\,(c), \mathit{getSymbol}\,(c))) \\ \mathit{isComp}\,(m,n) & = & \mathit{isComp}\,(\mathit{rest}\,(m), n) \wedge \mathit{eq}\,(\mathit{last}\,(m), \mathit{step}\,(\mathit{last}\,(\mathit{rest}\,(m)))) \\ \mathit{halted}\,(c) & = & \mathit{eq}\,(\mathit{digit}(\log(b \sim 2, c \sim 1) \sim 2, c, b), E(h)) \\ \mathit{comp}\,(n) & = & \mu m[\mathit{iscomp}\,(m,n) \wedge \mathit{halted}\,(\mathit{last}\,(m))] \end{array}$$

- $\mathit{getState}\,(c)$ devuelve $E(q)$ a partir de una configuración donde el estado es q;

- *getSymbol* (c) devuelve el símbolo donde está la cabeza de M_f en c, que es la cifra anterior a la que representa el estado;
- *deltaState* (st, sy) y *deltaSymbol* (st, sy) son las funciones que aplican la función δ de M_f a un estado y un símbolo, y devuelven el nuevo estado y el nuevo símbolo o acción;
- *doStep*(c, st, sy) calcula la nueva configuración que resulta de aplicar la acción sy (escribir un símbolo o moverse) y cambiar el estado a st;
- *step* (c) calcula la nueva configuración que resulta de c y de la δ de M_f;
- *rest* (m) quita la última configuración de m;
- *isComp* (m, n) es una función primitiva (un predicado) que dice si m es una secuencia de configuraciones de M_f que deriva de una computación a partir de $(s, \triangleright\_n')$ (donde n' es la representación binaria de n); notamos que la definición dada no es una definición correcta de primitiva recursiva;
- *halted* (c) dice si la configuración c es final;
- finalmente, *comp* (n) encuentra el primer m que es una secuencia correcta de configuraciones y que termina con una configuración final; hay que notar que la minimización usada aquí es la única minimización verdaderamente necesaria en toda esta construcción.

Definiendo correctamente todas las funciones, el teorema queda demostrado: $f(n) = num\,(output\,(last\,(comp\,(n))))$ es μ-recursiva y calcula lo mismo que M_f.

Capítulo 10

Jueves 04/11/10 (2 horas)

10.1 Historia de la tesis de Church (años 30)

(Para un resumen razonable de la cuestión, consulta
`http://en.wikipedia.org/wiki/History_of_the_Church-Turing_thesis`)

1. Peano 1889: Basándose en un trabajo de Dedekind, presenta nueve axiomas de la aritmética;
2. Hilbert 1900: El segundo y el décimo problema de Hilbert's introducen el Entscheidungsproblem (el "problema de la decisión");
3. 3 Hilbert 1928: Otra formulación de los tres problemas que derivan del segundo y del décimo;
4. Ackermann et al. 1927-1928: Hay funciones "sencillas" (y totales) que no son primitivas recursivas;
5. Gödel 1930-1931: No se puede demostrar la consistencia de un sistema formal dentro del mismo sistema;
6. Gödel 1934: Modifica uno de los candidatos para representar el concepto de "computación efectiva": ya no "recursión primitiva", sino "recursión general";
7. Kleene 1935;
8. Church 1936: Define "efectivamente calculable" como algo que la recursión general y el λ-cálculo hacen; no todos están de acuerdo;
9. Post 1936: La idea de "computación efectiva" de Church es derivada de un razonamiento inductivo, así que se puede expresar como una "ley de la naturaleza" y no como una definición;
10. Turing 1936: Máquinas que calculan como modelo de hombres que calculan;

11. Turing 1939: "Computación efectiva" es lo mismo que "computation con MdT" y al revés;
12. Rosser 1939: Equivalencia entre recursión, λ-cálculo y computación con Máquinas de Turing;
13. Kleene 1943: Formulación de la Tesis de Church como Tesis I, y propuesta de usarla como definición de "efectivamente calculable";
14. Kleene 1952: La primera formulación formal de la Tesis de Church y de la Tesis de Turing, y su identificación;
15. Gödel 1963: Sólo las MdT son posibles candidatas para representar el concepto de "efectivamente calculable";
16. Gandy 1980: "Computación por medio de máquinas" como proceso discreto, determinista y limitado en su alcanze por la velocidad de la luz ("efecto local");
17. Soare 1995;
18. Breger 2000: el problema de los "axiomas implicitos" en un sistema axiomático;
19. Sieg 2000: discute las "definiciones axiomáticas".

Tesis de Church-Turing (según Turing): Cualquier cosa se pueda calcular es Turing-computable.

Tesis de Church-Turing (según Church): Cualquier cosa que se pueda efectivamente calcular es recursiva.

Turing-equivalencia Los formalismos que calculan esta (la misma) clase de funciones se dicen Turing-equivalentes: las MdT se usan como referencia, y los demás formalismos se comparan con ellas.

10.2 La fuerza de la tesis

De la Sec. 3.7 del libro de Cutland **??**.

- diversos formalismos, introducidos y estudiados de forma independiente, llevaron a la misma clase de funciones;
- nadie ha encontrado hasta ahora una función que se pueda aceptar informalmente como calculable y que no se pueda calcular con un máquina de Turing.

Capítulo 11

Jueves 11/11/10 (2 horas)

11.1 Tareas indecidibles

- un posible punto de vista al hilo de la Tesis de Church: las MdT *que terminan para todo input* corresponden a la idea intuitiva de "algoritmo"
- si pensamos en las MdT como máquinas que aceptan lenguajes:
  - toda MdT tiene una representación finita como una cadena de finita símbolos de un alfabeto finito;
  - sabemos que las cadenas finitas de un alfabeto finito son enumerables;
  - entonces, existen un número enumerable de MdT (totales o también parciales)...
  - ...y de funciones μ-recursivas, y de programas *while*, etc.;
  - pero tambieén sabemos que los lenguajes con un alfabeto finito (siendo subconjuntos del conjuntos de las cadenas finitas de tal alfabeto) son más que enumerables (*nota: estamos hablando a la vez de "cadenas finitas de un alfabeto finito" para hablar de la descripción de una MdT, y para hablar los elementos del lenguaje que una MdT quiere decidir; no hay que mezclar los dos usos*);
  - conclusión: ¡las MdT no pueden decidir que un número muy pequeño de lenguajes!

Esto que acabamos de ver es muy interesante; pero es mucho más interesante **encontrar tareas que no se pueden cumplir dentro de un formalismo** (las MdT): tareas *indecidibles*.

Para hacer esto, vamos a usar una capacidad muy importante de las MdT: una especie de *introspección*. Es decir, a una máquina de Turing se le puede dar la descripción de otra máquina para que la ejecute. La MdT que es capaz de hacer esto se llama *Máquina Universal*. Nuestro objetivo es demostrar cosas usando un argumento diagonal aplicado a la Máquina Universal que recibe como input su propia descripción.

11.2 La Máquina universal

- Hasta ahora, las MdT han sido básicamente piezas de *hardware* que cumplen una tarea concreta;
- es decir, no son *programables*;
- pero queremos considerar las MdT también como *software*;
- queremos una MdT *genérica* que pueda *simular* todas las otras;
- lo mismo que un ordenador puede ejecutar cualquier tarea a través de un programa;
- en nuestro caso, el programa será la descripción de una MdT;
- es decir, el formalismo que expresa las MdT es un lenguaje de programación;
- el punto es que estos programas pueden ser interpretados por "otro programa": la MdT genérica o *universal* U;
- un programa que interpreta otro programa escrito en el mismo lenguaje no es una idea nueva: véase los programas de *bootstrap*, o un intérprete para ML escrito en ML.

Tenemos que definir el *lenguaje* para describir MdT: el lenguaje cuyas cadenas de símbolos sean descripciones de MdT. Lo primero: nos damos cuenta que no existe un alfabeto finito cuyos símbolos puedan representar todos los estados y todos los símbolos de todas las MdT. Por lo tanto, estados y símbolos de una máquina M se representan con cadenas de símbolos de U.

- sea $M = (K, \Sigma_M, \delta, s, H)$;
- la MdT U que buscamos tendrá alfabeto Σ_U;
- un estado de M se representa con $\{q\}\{0,1\}^*$, es decir, el símbolo q seguido por un número binario (Σ_U tiene que incluir q, 0 y 1);
- un símbolo de Σ_M se representa como $\{a\}\{0,1\}^*$ (entonces, $a \in \Sigma_U$);
- las cadenas binarias para los estados tendrán longitud i donde i es el número más pequeño tal que $2^i \geq |K|$; el estado inicial se representa siempre como $q0^i$;
- las cadenas binarias para los símbolos de Σ_M tendrán longitud j donde j es el número más pequeño tal que $2^j \geq |\Sigma_M| + 2$; el "más 2" es para incluir $\leftarrow$ y $\rightarrow$;
- por convención

- $\llcorner$ corresponde a $a0^{j-2}00$ (es decir, todos 0 hasta los últimos dos símbolos que también son 0);
- $\triangleright$ corresponde a $a0^{j-2}01$;
- $\leftarrow$ corresponde a $a0^{j-2}10$;
- $\rightarrow$ corresponde a $a0^{j-2}11$.

Llamamos a "M" la representación de M en el alfabeto Σ_U; esta representación básicamente consiste de la δ de M.

- "M" es una secuencia de cadenas (*tuplas*) de la forma (q, a, p, b) donde q y p son representaciones de estados y a es la representación de un símbolo, y b es la representación de un símbolo o de $\leftarrow$ o $\rightarrow$ (nota que Σ_U tiene que contener "(", ")" y ",");
- cada tupla (q, a, p, b) describe el hecho que $\delta(q, a) = (p, b)$;
- las tuplas están ordenadas lexicográficamente.

Así se puede representar cualquier MdT. Notamos que con la representación de δ y las convenciones, se puede saber todo de M: los estados K, los símbolos Σ, el estado inicial y los estados finales:

- aunque no se conoce los nombres concretos de los estados de M, se sabe cuántos son porque se puede ver los que aparecen en alguna tupla de δ;
- también para los símbolos: se puede ver los que aparecen en δ; además, de algunos se sabe el significado por convención (esto es importante sobre todo para $\leftarrow$ y $\rightarrow$);
- el estado inicial es $q0^i$ por convención;
- los estados finales de M serán los h por los que no existe en "M" ninguna tupla (h, a, p, b).

Ejemplo 11.1 *Sea* $M = (\{s, q, h\}, \{\llcorner, \triangleright, a\}, \delta, s, \{h\})$ *y* δ *definida como*

estado	*símbolo*	δ
s	a	$(q, \llcorner)$
s	$\llcorner$	$(h, \llcorner)$
s	$\triangleright$	$(s, \rightarrow)$
q	a	(s, a)
q	$\llcorner$	$(s, \rightarrow)$
q	$\triangleright$	$(q, \rightarrow)$

En este caso, $i = 1$ *y* $j = 3$*. Entonces símbolos y estados se representan como*

estado/símbolo	*representación*
s	$q00$
q	$q01$
h	$q10$
$\llcorner$	$a000$
$\triangleright$	$a001$
$\leftarrow$	$a010$
$\rightarrow$	$a011$
a	$a100$

Por ejemplo, la cadena $\triangleright aa\llcorner a$ *se representa como*

$$\text{“}\triangleright aa\llcorner a\text{”} = a001a100a100a000a100$$

Y la representación “M” de M es

$$\begin{aligned}\text{“}M\text{”} = \quad & (q00, a100, q01, a000), (q00, a000, q10, a000), (q00, a001, q00, a011), \\ & (q01, a100, q00, a011), (q01, a000, q00, a011), (q01, a001, q01, a011)\end{aligned}$$

Ahora podemos definir la *Máquina de Turing Universal* U, que usa las codificación de las otras MdT y las ejecuta sobre un input w. La propiedad que se desea de U es: *U para con el input “M” “w” si y sólo si M para con w.*

Vamos a definir U como una MdT con tres cintas (que, ya los sabemos, se puede simular por medio de una MdT con una cinta).

- la primera cinta recibe el input “M” “w”, y se usa para las computaciones de M;
- la segunda cinta contiene la codificación de M;
- la tercera cinta contiene el estado actual de M.

Las primeras operaciones de U son

- mover “M” de la primera a la segunda cinta;
- mover “w” al principio de la primera cinta, después de haber escrito la representación de $\triangleright\llcorner$; así lo que contiene la cinta es la representación de $\triangleright\llcorner w$;
- la cabeza de la primera cinta se posiciona en el primer símbolo de la representación del primer $\llcorner$;
- escribir el estado inicial en la tercera cinta (notamos que su representación es siempre $q0^i$, e i se puede calcular por medio de una “inspección” de la codificación de M (U puede hacer este trabajo);
- la segunda y tercera cabeza se posicionan, por ejemplo, al inicio de sus cintas (este detalle no es importante).

Después, U simula un paso de M haciendo lo siguiente:

- lee y decodifica el símbolo σ de Σ_M en la primera cinta (la cabeza ya estaba en la posición correcta);
- lee el estado q de M en la tercera cinta;
- a la vez (no puede "memorizar" lo que ha visto en sus estados, sino que tiene que comparar el contenido de la primera y tercera cinta con el de la segunda) busca la pareja (q, σ) en las tuplas de la segunda cinta:
  - si la encuentra, copia el estado resultante a la tercera cinta y escribe el símbolo resultante en la primera, o se mueve según el símbolo que encontró;
  - si no la encuentra (es decir, si el estado de M era final) para en un estado final.

Capítulo 12

Martes 23/11/10 (1 hora)

12.1 El problema de la parada

La Máquina Universal nos permite enfrentarnos con uno de los resultados fundamentales de Teoría de la Computabilidad: el *problema de la parada* (*halting problem*), propuesto por Turing [].

La novedad de U con respecto a lo visto hasta ahora es que U es una MdT que recibe como input (representaciones de) otras MdT.

Con el lenguaje *while* Para introducir el problema, consideremos un lenguaje de programación más familiar (por ejemplo, el lenguaje *while*) donde suponemos tener una representación (con un múmero intero, ya que no tenemos otra cosa) de programas que puede ser el input de otro programa. Es decir, la llamada a procedimiento `p(q)` es en realidad el procedimiento `p` al que se le da como argumento la representación del procedimiento `q`.

Entonces, podemos pensar en un programa `halts(p,n)` que, para todo input `p`, dice si es la representación de alguún programa P que termina con el input n o no. Es decir, `halts` contesta 1 si $P(n)$ termina, y 0 si $P(n)$ no termina.

Ahora, pensemos en otro programa `diagonal(x)`:

```
1 diagonal(x) {
2   while (halts(x,x)) {}
3 }
```

Esto significa que el programa termina si y sólo si el programa representado por `x` **no** termina con el input `x`.

La siguiente pregunta es: **¿`diagonal(diagonal)` termina?**. Si termina, entonces `halts(diagonal,diagonal)` devuelve 0, es decir, `diagonal` no termina. Si no termina, entonces `halts(diagonal,diagonal)` devuelve 1, es decir, `diagonal` termina. Es decir `diagonal(diagonal)` termina si y sólo si no termina.

Es claramente una contradicción; por lo tanto, el hipótesis del que partimos es falso. ¿Cuál es este hipótesis? Que `halts` existe. Por lo tanto,

no existe ningún programa que, para todo programa P y todo input n, diga siempre (y correctamente) si $P(n)$ termina o no

FiXme Fatal: Was it him? Check cite

Con las Máquinas de Turing Lo que vimos acerca de la Máquina Universal nos permite formalizar este argumento, casi como lo hizo Turing [17].

Sea H el lenguaje definido como

$$H = \{"Mw" \mid \text{la máquina } M \text{ termina con el input } w\}$$

Este lenguaje es semidecidible, o recursivamente enumerable, porque la MdT universal es justamente la máquina que lo semidecide. El resultado de U cuando se la da un input in es

- si in describe M y w, y M termina en w, entonces U termina en el estado y de aceptación y acepta el input; o
- si in describe M y w, y M no termina en w, entonces U tampoco termina (y no acepta el input); o
- si in no es una descripción correcta de una MdT y de su input, entonces U termina el estado n de rechazo y no acepta el input.

Nos preguntamos si H, además de ser recursivamente enumerable, es también recursive (es decir, existe una MdT que lo decide).

La primera observación es que, dado el lenguaje H_1:

$$H_1 = \{"M" \mid \text{la máquina } M \text{ termina con el input } "M"\}$$

- si H es recursivo entonces H_1 también lo es; de hecho, si tuvieramos una MdT M_H que decide H, podríamos construir una M_{H1} que decide H_1:
  - dado el input $"M"$, M_{H1} duplica el input en la cinta (obteniendo $"MM"$)
  - M_{H1} llama a M_H para que acepte o rechaze $"MM"$;
- por lo tanto, si demostramos que H_1 no es recursivo, tampoco H lo será.

La segunda observación es que, dado el lenguaje $\overline{H_1}$ complementario a H_1:

$$\overline{H_1} = \{w \mid (w \text{ no describe un MdT}) \text{ o } (w \text{ describe una MdT } M \text{ que no termina con } "M"\}$$

Si H_1 fuera recursivo, entonces $\overline{H_1}$ también lo sería, ya que los lenguaje recursivos son cerrados por complementación.

Entonces, la recursividad de H implica la recursividad de $\overline{H_1}$, por lo que, si demostramos que $\overline{H_1}$ no es recursivo, demostramos que H tampoco lo es.

De hecho, podemos demostrar que $\overline{H_1}$ no es ni siquiera recursivamente enumerable.

1. Supongamos que la MdT M^* semidecida $\overline{H_1}$: entonces, para todo w, $w \in \overline{H_1}$ si y sólo si M^* lo acepta. Pero ¿pertenece "M^*" a $\overline{H_1}$? Si pertenece, entonces, por la definición del conjunto, la MdT M^* no termina en el input "M^*", que es equivalente a decir que M^* no se acepta a sí misma, es decir, que $w \notin \overline{H_1}$.

2. En cambio, si $w \notin \overline{H_1}$, entonces M^* no acepta "M^*".

   - Sin perder generalidad, podemos suponer que M^* tenga sólo un estado final y y tenga sólo dos resultados:
     - termina en y (acepta); o
     - no termina (rechaza).

   Si no acepta "M^*" entonces no termina; por lo tanto, "M^*" $\in \overline{H_1}$.

Hemos demostrado "M^*" $\in \overline{H_1} \Leftrightarrow$ "M^*" $\notin \overline{H_1}$, que es claramente una contradicción. Por lo tanto, no puede existir esta MdT M^* que semidecide $\overline{H_1}$, y este lenguaje se demuestra no ser recursivamente enumerable.

El resultado final es que hemos encontrado un conjunto H que es recursivamente enumerable per no es recursivo.

Ya sabíamos que existen lenguajes que no son recursivos; pero éste es un ejemplo concreto, y sobre todo interesante. H es interesante porque veremos que saber si un programa/MdT termina dado cierto input es un área de investigacioón relevante en la informática actual.

Capítulo 13

Jueves 25/11/10 (2 horas)

13.1 La computabilidad en "On Computable Numbers, with an Application to the Entscheidungsproblem" [17]

Este artículo de enorme importancia contiene, entre otras cosas, la formalización de las Máquinas (de Turing) y muchas discusiones sobre asuntos fundamentales de la teoría de la computabilidad.

- Lo "calculable" según Turing: números calculables
- La idea de una computador: un hombre que computa
- Máquinas para calcular
- Máquina *circular* y máquina *circle-free*
- La máquina universal
- Los números calculables son enumerables
- El problema de la parada en versión original
- Los números calculables y el concepto de cálculo
- Ejemplos de clases de números calculables
- Aplicación al Entscheidungsproblem
- Relación entre máquinas ("computable") y λ-cálculo ("effectively computable")

Curiosamente, el artículo contiene también unos fallos (en una demostración, y ne ciertas afirmaciones, que, sin ser falsas, necesitaban unos cambios), de cuya corrección el mismo Turing se hizo cargo en un anexo.

13.2 Problemas indecidibles con las MdTs

Dos puntos de vista:

- si aceptamos el principio que *todo algoritmo se puede expresar con una MdT que termina en todo input*, entonces la conclusión del problema de la parada es que no existe un algoritmo que decida para toda M y todo w si M acepta w;
- si aceptamos el principio que *todo algoritmo se puede expresar con una MdT* (aunque ésta no termine siempre), entonces la conclusión es que un algoritmo que decida para toda M y todo w si M acepta w no puede dar una respuesta siempre.

Elegimos el primer punto de vista
citelewisPapadimitriou; problemas por los que no existe un algoritmo "total" se llaman *indecidibles* o *insolubles*. El más importante, lo acabamos de ver, es el problema de la parada.

En lugar de buscar problemas y demostrar que son indecidibles aplicando algún proceso de diagonalización, vamos a uasr el concepto de *reducción* para demostrar que un problema es indecidible a partir de otro problema del que ya se conoce la indecidibilidad.

Definición 13.1 *Sean $L_1 \subseteq \Sigma^*$ y $L_2 \subseteq \Sigma^*$ dos lenguajes. Una* reducción *de L_1 a L_2 es una función recursiva (total) $\tau : \Sigma^* \mapsto \Sigma^*$ tal que $x \in L_1$ si y sólo si $\tau(x) \in L_2$.*

Es muy importante que τ sea recursiva total. También es importante la dirección en la que se reduce.

Para demostrar que un lenguaje L_2 *no* es recursivo, hay que

- encontrar un lenguaje L_1 del que ya sabemos que no es recursivo; y
- encontrar una reducción de L_1 a L_2.

Haciéndolo al revés no obtendríamos ningún resultado útil.

Teorema 13.1 *Si L_1 es no recursivo y existe una reducción de L_1 a L_2 entonces L_2 tampoco es recursivo.*

Demostración. Si L_2 fuera recursivo, entonces existiría una MdT M_2 que lo decide. Si T es la MdT que calcula la reducción τ, entonces TM_2 es la MdT que decide L_1, pero esto es imposible por el hipótesis que L_1 no es recursivo. □

Ejercicio 13.1 *Hay una serie de problemas sobre Máquinas de Turing que se conocen como indecidibles. Tratar de demostrar su indecidibilidad relacionándolo cada uno con un lenguaje no recursivo L_2 al que se pueda reducir otro lenguaje L_1 del que se conozca ya su no-recursividad (en el primer ejemplo, habrá que usar H).*

- *dada M, ¿termina M en la cinta vacía?*
- *dada M, ¿hay un input por el que M termina?*
- *dada M, ¿termina M en todo input?*
- *dadas M_1 y M_2, ¿terminan en los mismos inputs?*
- *dada M, el lenguaje que semidecide ¿es regular? ¿es libre de contexto? ¿es recursivo?*

Ejercicio 13.2 *Existe una MdT M por la que este problema es indecidible: "dado w, ¿termina M en w?". ¿Cuál es esta máquina?*

Ejercicio 13.3 *Se dice que una MdT M* usa *k casillas para un input w sii existe una configuración $(q, \triangleright u\underline{a}v)$ tal que $(s, \triangleright \triangleright \underline{\sqcup}w) \vdash^* (q, \triangleright u\underline{a}v)$ (es decir, que M alcanza dicha configuración), y la longitud de uav es por lo menos k.*

1. *Demostrar que el siguiente problema es* soluble*: dada una MdT M, un input w y un número k, ¿M usa k casillas para el input w?*
2. *Supongamos que $f : \mathbb{N} \mapsto \mathbb{N}$ sea recursiva. Demostrar que el siguiente problema es soluble: dada una MdT M y un input w, ¿M usa $f(|w|)$ casillas para el input w?*
3. *Demostrar que el siguiente problema es* insoluble*: dada una MdT M y un input w, ¿existe $k \geq 0$ tal que M* no *usa k casillas para el input w (es decir, tal que M una cantidad finita de cinta)?*

Una curiosidad: la conjectura de Goldbach. Existe un algoritmo para solucionar el problema, y sin embargo se trata de una conjectura por la que no tenemos todavía respuesta. La respuesta es claramente calculable: ¡es 1 (verdadero) o 0 (falso)! Entonces, el algoritmo correcto es, por ejemplo, la función primitiva recursiva constante "1" o la función primitiva recursiva constante "0". El problema es que *no sabemos cuál de los dos algoritmos es el correcto...*

Capítulo 14

Martes 30/11/10 (1 hora)

Discusión sobre los ejercicios 13.1 y 13.3.

Capítulo 15

Jueves 02/12/10 (2 horas)

Más discusión sobre los ejercicios propuestos.

Otros ejercicios.

Ejercicio 15.1 *¿Cuáles de estos problemas son solubles, y cuáles no lo son? Justificar la respuesta.*

1. *dados M y w, y un estado q, decidir si $M(w)$ alcanza en algún momento el estado q;*
2. *dados M y w, y dos estado p y q, decidir si $M(w)$ alcanza una configuración con p tal que la siguiente configuración incluye el estado q;*
3. *dados M y q, decidir si M alcanza q para algún input;*
4. *dada M y un símbolo a, decidir si $M(\varepsilon)$ (cinta vacía) escribirá a en la cinta alguna vez;*
5. *dada M, decidir si $M(\varepsilon)$ (cinta vacía) escribirá un símbolo que no sea $\sqcup$ en la cinta alguna vez;*
6. *dados M y w, decidir si $M(w)$ moverá su cabeza hacia izquierda alguna vez;*
7. *dadas M_1 y M_2, determinar si M_1 semidecide el complemento del lenguaje que M_2 semidecide;*
8. *dadas M_1 y M_2, determinar si hay un input para el que las dos terminan;*
9. *dada M, determinar si el lenguaje que semidecide es finito.*

Capítulo 16

Jueves 09/12/10 (2 horas)

16.1 Resultados clásicos

Supongamos tener una *codificación* de representaciones de Máquinas de Turing a números naturales. Notamos que vale una codificación cualquiera, y no hace falta que sea surjectiva (el único requisito es que sea inyectiva). Sea M_i la MdT tal que la codificación de "M" es i. Sea φ_i la función de $\mathbb{N}$ a $\mathbb{N}$ que M_i calcula. También se puede codificar una computación, que no es otra cosa que una secuencia de configuraciones. Codificación y decodificación se pueden implementar como funciones *primitivas recursivas*.

Se dice que una función calculable ϕ tiene un índice i si M_i calcula ϕ.

Teorema 16.1 *Cada función calculable ϕ tiene un número infinito (enumerable) de índices.*

Demostración. Si ϕ es calculable entonces habrá una MdT M_i que la calcula, por un cierto i (es decir, $\phi = \varphi_i$). También se puede construir una secuencia enumerable de MdTs que calculan la misma función añadiendo a su δ reglas que en realidad no cambian nada de su comportamiento

Por ejemplo, la primera de esta secuencia será como M_i pero con un estado q_1 añadido y unas nuevas reglas en δ: $\delta(q_1, \sigma) = (q_1, \sigma)$ para todo σ (notamos que nunca se alcanzaría el estado q_1). La segunda MdT tendrá otro nuevo estado q_2 y las correspondientes reglas (tampoco q_2 se alcanzaría). El proceso se puede iterar sin límite, por lo que se puede construir una secuencia infinta (numerable) de máquinas que hacen todas lo mismo. □

Teorema 16.2 (Forma normal) *Para toda φ_i calculable existen un predicado $T(i, x, y)$ y una función $U(y)$ primitivos recursivos y tales que $\varphi_i(x) = U(\mu y.T(i, x, y))$.*

Demostración. $T(i, x, y)$ sería el predicado que devuelve 1 si y es la codificación de una computación terminante de M_i con input x: a partir de i e y se

puede recuperar (decodificando) M_i y la computación $c_0c_1...c_n$, y luego comprobar si c_0 es $(s, \triangleright\_x)$, y se extrae z de $c_n = (h, \triangleright\_z)$. Si todo esto se puede hacer entonces $U(y) = z$ (es decir, la función que saca z de la última configuración). T y U son primitivas recursivas porque la codificación lo es. □

Este teorema nos dice que cada función calculable necesita una sola aplicación del operador μ a funciones primitivas recursivas. Pero también que, en teoría, cada programa en el lenguaje *while* es equivalente a uno donde se usa una vez el comando **while** y dos veces el comando **for**.

Vamos a ver el *teorema del parámetro*, o *teorema s-m-n*, en el caso especial $m = n = 1$.

Teorema 16.3 (s-1-1) *Existe una función calculable total y biunívoca s_1^1 con dos argumentos, tal que, para todo x, y, z:*

$$\varphi_{s_1^1(x,y)}(z) = \varphi_x(y, z)$$

Intuitivamente la máquina $M_{s_1^1(x,y)}$ trabaja sólo con z, mientras M_x tiene dos argumentos. Entonces podemos decir que y es un parámetro *de M_x.*

Por ejemplo, si $\varphi_x(y, z)$ es la función $y+f(z)$ por una f cualquiera, a partir de x y de, por ejemplo, el número 2 obtengo es modo efectivo, calculando $s_1^1(x, 2)$, el índice de la máquina que calcula $2 + f(z)$.

El teorema del parámetro es importante porque es la base de la técnica llamada *evaluación parcial*: partiendo de un programa general, y dado un input (parcial), se obtiene un programa especializado y probablemente más eficiente. Notamos que, si lo que queremos es eficiencia, esta operación no es nada fácil (no basta, en general, con sustituir variables con constantes), y el teorema dice sólo que se puede hacer, no cómo se hace.

La versión general del teorema es:

Teorema 16.4 (s-m-n) *Existe una función calculable total y biunívoca s_n^m con $m + 1$ argumentos, tal que, para todo x, $y_1..y_m$, $z_1..z_n$:*

$$\varphi_{s_n^m(x,y_1..y_m)}(z_1..z_n) = \varphi_x(y_1..y_m, z_1..z_n)$$

Un último teorema, muy elegante, se demuestra con el $s - m - n$.

Teorema 16.5 (Recursión II de Kleene) *Para toda función f calculable total existe un n tal que $\varphi_n = \varphi_{f(n)}$. Tal n se dice* punto fijo *de f.*

Demostración. Definimos la siguiente función calculable parcial:

$$\psi(v, z) = \varphi_{d(v)}(z) = \begin{cases} \varphi_{\varphi_v(v)}(z) & \text{si } \varphi_v(v) \text{ termina} \\ \text{indefinida} & \text{en otro caso} \end{cases}$$

Por el teorema s-1-1, d es total y biunívoca (y no depende de f). Ahora, dada la f del enunciado del teorema, hay un índice v tal que $\varphi_v(x) = f(d(x))$ para todo

x, y este índice existe porque f y d son las dos calculables y totales. Por lo tanto φ_v es también total, y entonces $\varphi_v(v)$ termina. De esto deriva que $\varphi_{d(v)}(z)$ es exactamente $\varphi_{\varphi_v(v)}(z)$, según la definición de arriba, para todo z (es decir, las dos funciones son iguales). Por lo tanto, sea $n = d(v)$:

$$\varphi_n = \varphi_{d(v)} = \varphi_{\varphi_v(v)} = \varphi_{f(d(v))} = \varphi_{f(n)}$$

que es el enunciado del teorema. Todos los $d(v)$ son puntos fijos de f. Es decir, f tiene infinitos puntos fijos. □

La función f es un *transformador de programas* que no cambia la semántica. Este teorema es la base de la semántica denotacional, de los programas recursivos, de la criptografía, etc.

Capítulo 17

Martes 14/12/10 (1 hora)

17.1 Conjuntos de índices y teorema de Rice

Definición 17.1 (Conjunto de índices) *$A \subseteq \mathbb{N}$ es un* conjunto de índices *si y sólo si, para todo x e y, si $x \in A$ y $\varphi_x = \varphi_y$ (es decir, si las MdTs cuyas codificaciones son x e y calculan la misma función calculable parcial), entonces $y \in A$.*

Si un índice está en un conjunto de índices A, entonces todas las máquinas que calculan la misma función están también en A.

Nos acordamos de que $H_1 = \{"M" \mid \text{la máquina } M \text{ termina para el input } "M"\}$, y escribimos el mismo lenguaje con las codificaciones:

$$H_1 = \{x \mid \text{la máquina cuya codificación es } x \text{ termina para el input } x\}$$

Teorema 17.1 *Sea A un conjunto de índices tal que $\emptyset \neq A \neq \mathbb{N}$. Entonces H_1 se reduce a A o a $\overline{A}$.*

Demostración. Sea i_0 un índice tal que φ_{i_0} es la función siempre indefinida. Supongamos que $i_0 \in \overline{A}$, y vamos a demostrar que H_1 se reduce a A. Si fuera que $i_0 \in A$, con una demostración muy parecida se demostraría que H_1 se reduce a $\overline{A}$. Dado que $A \neq \emptyset$ por hipótesis, existe un índice i_1 en A, y $\varphi_{i_0} \neq \varphi_{i_1}$ porque A es un conjunto de índices.

Sea $\varphi_i(x, y)$ la función parcial definida como

$$\varphi_i(x,y) = \begin{cases} \varphi_{i_1}(y) & \text{si } x \in H_1 \\ \varphi_{i_0}(y) = \text{indefinida} & \text{si } x \notin H_1 \end{cases}$$

Por el teorema s-m-n, existe una función s_1^1 *total y biunívoca* tal que $\varphi_i(x,y) = \varphi_{s_1^1(i,x)}(y)$. Sea $f = \lambda x.s_1^1(i,x)$. Entonces, la definición se puede reescribir como

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_{i_1}(y) & \text{si } x \in H_1 \\ \varphi_{i_0}(y) = \text{indefinida} & \text{si } x \notin H_1 \end{cases}$$

donde la cosa importante es que la condición (si x está o no está en H_1) no depende del parámetro y de $\varphi_{f(x)}$.

Entonces vemos que esta f es justamente la función π que identifica la reducción de H_1 a A; de hecho, usando la propiedad de A de ser un conjunto de índices:

$$\begin{array}{rcccl} x \in H_1 & \Rightarrow & \varphi_{f(x)} = \varphi_{i_1} & \Rightarrow & f(x) \in A \\ x \notin H_1 & \Rightarrow & \varphi_{f(x)} = \varphi_{i_0} & \Rightarrow & f(x) \notin A \end{array}$$

Y el teorema queda demostrado. □

Este teorema dice que todo conjunto de índices es no-recursivo, a no ser que sea uno de los conjuntos triviales $\emptyset$ o $\mathbb{N}$.

El siguiente corolario pone límites estrictos a las propiedades demostrables sobre las funciones calculables.

Teorema 17.2 (Rice) *Sea $\mathcal{P}$ un conjunto de funciones calculables parciales (nota: de funciones, no de índices). El conjunto (éste sí, de índices) $L_{\mathcal{P}} = \{n|\varphi_n \in \mathcal{P}\}$ es recursivo si y sólo si es $\emptyset$ o es $\mathbb{N}$.*

En resumidas cuentas, el teorema de Rice dice que cualquier propiedad no trivial $\mathcal{P}$ *de lenguajes recursivamente enumerables* es indecidible. Una propiedad trivial de lenguajes recursivamente enumerables es una que (i) cumplen todos los lenguajes o (ii) no cumple ninguno.

Notamos que, para que tenga sentido preguntarnos si $L_{\mathcal{P}}$ es $\mathbb{N}$, es necesario que la codificación de una MdT sea suryectiva: de este modo, a todo índice n le corresponde una φ_n y una M_n. Si la codificación no fuera suryectiva $L_{\mathcal{P}}$ podría ser decidible sin ser $\mathbb{N}$ si fuera $\{n \mid n$ es la codificación de una MdT$\}$. Ahora, las codificación sencillas que vimos no son suryectivas, pero no es difícil encontrar codificaciones suryectivas que sigan siendo primitivas recursivas.

Capítulo 18

Jueves 16/12/10 (2 horas)

18.1 Dos definiciones de lenguajes RE

Sea $L(M)$ el lenguaje aceptado (semidecidido) por la MdT M, es decir, el conjunto de inputs por los que M termina en un estado de aceptación (para los demás inputs, M puede rechazar o bien no terminar).

En la literatura se encuentran al menos dos definición de lenguajes recursivamente enumerables (RE):

1. L es RE si existe una MdT que lo *semidecide*; es decir, si $L = L(M')$ por alguna M';

2. L es RE si existe una MdT M'' que lo *enumera*; es decir, M'' escribe en la cinta todos y sólo los elementos de L.

La segunda definición es menos intuitiva: estamos acostumbrados a que las MdTs dependan de un input, pero M'' no lo tiene (siempre hace lo mismo). Sin embargo, esta definición es más parecida a la que dio Turing de *computable number* [17].

Queremos demostrar que las dos definiciones son equivalentes.

Lema 18.1 *Dado un lenguaje L, existe una máquina de Turing M' que lo semidecide si y sólo si existe una máquina de Turing M'' que lo enumera.*

Demostración. Si existe M' tal que $L = L(M')$, entonces $x \in L$ sii $M'(x)$ termina en un estado de aceptación. Entonces M'' se puede construir así:

- en la fase 1 ejecuta el primer paso de $M'(1)$ (M' con el input 1, según cierta representación de 1);

- en la fase 2 ejecuta el primer paso de $M'(2)$ y el segundo de $M'(1)$;

- en la fase 3 ejecuta el primer paso de $M'(3)$, el segundo de $M'(2)$ y el tercero de $M'(1)$;

- en la fase n ejecuta el primer paso de $M'(n)$, el segundo de $M'(n-1)$, ..., y el paso n de $M'(1)$;
- si una de las computaciones termina, entonces no hará falta ejecutar los siguientes pasos.

De este modo, M'' gestiona un número ilimitado de computaciones, cada una con un número ilimitado de pasos. ¿Cómo puede M'' hacer todo a la vez? Básicamente usando (en principio) cuatro cintas:

- en la primera cinta, inicialmente vacía, escribe progresivamente L;
- en la segunda, de sola lectura, está escrita la descripción "M'" de M';
- en la tercera M'' escribe los estados de M' en cada una de las n computaciones activas en la fase n, además de otra información como el mismo número n, o la computación que tiene que dar el siguiente paso;
- en la cuarta escribe los resultados parciales de todas las computaciones: en la fase n, estarán activas n computaciones, por lo que es suficiente dar a cada computación una parte de la cinta (con separadores oportunos, y teniendo en cuenta que ampliar el espacio de una computación requiere desplazar toda la parte siguiente de la cinta hacia la derecha);

Cuando una computación $M'(x)$ termina en un estado de aceptación, M'' escribe x al final de la primera cinta y sigue con las demás computaciones; si $M'(x)$ termina en un estado de rechazo, se puede borrar la computación de todas las cintas y seguir con las demás. Se puede ver que M'' enumera el lenguaje L. Su estructura es tal que, a través del método de *dove-tailing* que ya hemos visto, ejecuta un número ilimitado de pasos de un número ilimitado de computaciones.

Por otro lado, hay que demostrar que si existe tal M'' que enumera L entonces existe una M' que semidecide L. Dado un input x, M' simplemente ejecuta M'' hasta que genere x, y, si lo hace, termina en un estado de aceptación. Si M'' termina sin haber generado x, entonces M' termina en un estado de rechazo. Si M'' no termina y nunca genera x, M' tampoco termina. En todo caso, M' semidecide L. □

18.2 El significado del teorema de Rice

El teorema de Rice dice que $L_{\mathcal{P}} = \{n | \varphi_n \in \mathcal{P}\}$ es recursivo si y sólo si es $\emptyset$ o es $\mathbb{N}$. Es decir, cualquier conjunto de índices que no sea trivial es indecidible como lenguaje. ¿Por qué $L_{\mathcal{P}}$ es un conjunto de índices? Porque si $n' \in L_{\mathcal{P}}$ entonces $\varphi_{n'} \in \mathcal{P}$; para todo n'' tal que $\varphi_{n''} = \varphi_{n'}$, tendrá que ser que $\varphi_{n''} \in \mathcal{P}$, porque, desde el punto de vista matemático, son la misma función. Pero entonces $n'' \in L_{\mathcal{P}}$ por definición.

Definición 18.1 (Propiedad sobre un conjunto) *Una* propiedad *sobre un conjunto X es (se puede representar como) un subconjunto Y del mismo; esto significa que un elemento x de X satisface la propiedad sii pertenece a Y. Por ejemplo, si $X = \mathbb{N}$, la propiedad "ser número par" se puede representar con el conjunto $Y \subseteq \mathbb{N}$ de los números pares. En el teorema de Rice, $\mathcal{P}$ se entiende como una* propiedad sobre las funciones calculables parciales*, siendo un conjunto de ellas.*

Además, $L_{\mathcal{P}}$ es una *propiedad sobre los lenguajes recursivamente enumerables*. La segunda observación es menos intuitiva, porque al fin y al cabo $L_{\mathcal{P}}$ es un conjunto de números (y de índices). ¿Por qué es una propiedad sobre languajes RE?

Porque preguntarse si n pertenece a $L_{\mathcal{P}}$ es igual, por definición, que preguntarse si φ_n está en (satisface) $\mathcal{P}$. Ahora, φ_n es una función de $\mathbb{N}$ a $\mathbb{N}$ (no cambiaría nada, lo sabemos, si el dominio fuera $\mathbb{N}^k$ por un cierto k, o Σ^* por un cierto Σ).

Sea $F_{\mathbb{N}}$ el conjunto de las funciones calculables parciales de $\mathbb{N}$ a $\mathbb{N}$, y $F_{0,1}$ el conjunto de las funciones calculables parciales de $\mathbb{N}$ a $\{0,1\}$. La cardinalidad de los dos conjuntos es la misma (notamos que son los dos *enumerables*, mientras que la funciones no necesariamnete calculables son más), por lo que existe una *correspondencia biunívoca* π' entre $F_{\mathbb{N}}$ y $F_{0,1}$.

Entonces, preguntarse si φ_n (función calculable parcial de $\mathbb{N}$ a $\mathbb{N}$) está en $\mathcal{P}$ es igual que preguntarse si $\pi'(\varphi_n)$ (función calculable parcial de $\mathbb{N}$ a $\{0,1\}$ que "corresponde" a φ_n) está en $\mathcal{P}'$, donde $\mathcal{P}'$ es $\mathcal{P}$ transformado según la correspondencia biunívoca: $\mathcal{P}' = \{\pi(\psi) | \psi \in \mathcal{P}\}$.

Además, a una función calculable parcial $\psi \in F_{0,1}$ corresponde uno y un solo lenguaje RE L (sobre los números naturales), porque

- existe una MdT M_ψ que calcula ψ (por ser calculable);
- M_ψ se puede fácilmente transformar en otra M_L tal que $M_L(x)$ acepta x si y sólo si $M_\psi(x) = 1$;
- entonces el lenguaje $L = L(M_L)$ que es semidecidido por M_L es el lenguaje correspondiente a ψ.

También, a cada L recursivamente enumerable le corresponde una sola ψ (se puede comprobar dando los mismos pasos al revés), así que existe una correspondencia biunívoca π'' entre $F_{0,1}$ y los lenguajes RE.

La pregunta inicial se reduce a preguntarse si el lenguaje $L = \pi''(\pi'(\varphi_n))$ correspondiente a la función $\pi'(\varphi_n)$ que corresponde a φ_n está en $\mathcal{P}''$, que es $\mathcal{P}'$ transformado según π'': $\mathcal{P}'' = \{\pi''(\psi) | \psi \in \mathcal{P}'\}$.

Resumiendo, $\varphi_n \in \mathcal{P}$ si y sólo si $\pi''(\pi'(\varphi_n)) \in \mathcal{P}''$, por lo que al final hemos transformado una pregunta sobre índices en una pregunta sobre lenguajes RE. Entonces, se puede ver $L_{\mathcal{P}}$ como una propiedad de lenguajes RE. Más formalmente:

Teorema 18.1 *Existe una función biunívoca total π de $F_{\mathbb{N}}$ al conjunto de lenguajes RE tal que, para toda propiedad $\mathcal{P}$ de funciones calculables parciales, satisface*

$$\forall n.(\varphi_n \in \mathcal{P} \Leftrightarrow \pi(\varphi_n) \in \pi(\mathcal{P})$$

Donde $\pi(\mathcal{P})$ es el conjunto $\{\pi(x)|x \in \mathcal{P}\}$.

Demostración. La función π es la composición de π' y π'' descritas arriba: para todo x, $\pi(x) = \pi''(\pi'(x))$. El resto de la demostración recorre simplemente el razonamiento expuesto en los párrafos anteriores. □

Existen varias versiones del teorema de Rice donde el conjunto que es indecidible (a no ser que él o su complemento sean el conjunto vacío) es, por ejemplo,

- el conjunto de "descripciones" $\{"M" \mid L(M) \in \mathcal{P}\}$, o bien el conjunto de índices (se puede comprobar que es un conjunto de índices según la definición) $\{n \mid L(M_n) \in \mathcal{P}\}$, según si la codificación es "numérica" o "simbólica";
- el conjunto $\{"M" \mid$ el lenguaje generado por M está en $\mathcal{P}\}$, o también $\{n \mid$ el lenguaje generado por M_n está en $\mathcal{P}\}$, según la codificación.

En estos casos, $\mathcal{P}$ aparece explícitamente como una propiedad sobre lenguejs RE. Todas las versiones son equivalentes.

Ejemplo 18.1 *Propiedades no decidibles de funciones calculables:*

- *si φ_x es la función constante 0; basta con coger $\mathcal{P} = \{\lambda y.0\}$ y ver que hay índices i', i'' tales que $i' \in L_{\mathcal{P}}$ e $i'' \notin L_{\mathcal{P}}$;*
- *si φ_x devuelve 0 para al menos un input;*
- *si φ_x no devuelve 0 para ningún input;*
- *si φ_x es constante;*
- *si φ_x es no-constante;*
- *si φ_x es total;*
- *si φ_x tiene dominio infinito;*

En muchos casos, los lenguajes asociados a estos problemas no son ni siquiera RE; por ejemplo, no se puede enumerar el conjunto de funciones calculables totales.

Ejemplo 18.2 *Propiedades no decidibles de lenguajes RE:*

- *que el lenguaje sea o no sea vacío: $\{"M" \mid L(M)$ es vacío$\}$;*
- *que el lenguaje sea o no sea finito: $\{"M" \mid L(M)$ es finito$\}$;*

- *que el lenguaje sea o no sea regular:* $\{"M" \mid L(M)$ *es regular*$\}$*;*
- *que sea o no sea libre de contexto:* $\{"M" \mid L(M)$ *es libre de contexto*$\}$.

No hay que pensar que todos los problemas sobre funciones calculables sean "víctimas" del teorema de Rice. Es importante entender que la propiedad $\mathcal{P}$ de la que habla el teorema es una propiedad *de las funciones*, no una propiedad de los índices, ni de funciones e índices a la vez. Así pues, el teorema de Rice no se aplica, por ejemplo, al problema (que, de todas formas, es también indecidible) de determinar, para una específica función total f y un x arbitrario, si φ_x es igual que $\varphi_{f(x)}$, ya que estaríamos hablando de una propiedad que tiene que ver con índices.

Entonces ¿qué nos dice el teorema? Siguiendo un poco más en la reformulación del teorema, está claro que lo que es indecidible es un conjunto (no trivial) de índices, es decir, de Máquinas de Turing. En otros términos, no existe un método general para decidir si una MdT satisface alguna propiedad *semántica* (de la función que calcula).

Pasando a un contexto más familiar, y observando que las MdTs son un formalismo de computación entre muchos, se puede decir que, en general, no existe un método para decidir si *un programa* satisface alguna propiedad *semántica* no trivial. Y lo más importante es que esto vale también para propiedades muy sencillas como calcular la función 0 constante. La propiedad es semántica porque dos funciones calculables iguales (a pesar de ser calculadas por programas distintos) son iguales según $\mathcal{P}$. Es decir, $\mathcal{P}$ no distingue entre programas distintos que calculen lo mismo (es decir, semánticamente equivalentes).

Ejemplo 18.3 *Si* $\mathcal{P} = \{\lambda y.0\}$*, el problema es decidir si un programa calcula o no esta función. Es razonable imaginar un* analizador *que, dado el programa* `return 0;`*, diga inmediatamente que la propiedad es satisfecha. También, podemos pensar que este mismo analizador devuelva una respuesta negativa para el programa* `return 1;`*. Sin embargo, hay muchos más programas que calculan la misma función (de hecho, cuando hablamos de MdTs e índices, sabemos que toda función calculable tiene infinitos iíndices, es decir, MdTs que la calculan; y también hay infinitos programas que calculan la misma función): pensemos en* `return f(x);` *donde* `f(x)` *implementa la función constante* 0 *ejecutando una serie de cálculos muy complicados cuyo resultado acaba siempre siendo "milagrosamente" 0. El teorema dice que nuestro analizador se equivocará para algunos de estos programas* difíciles *(u* ofuscados*).*

Nota: el tema de los programas *ofuscados* será de actualidad en la segunda parte del curso.

Capítulo 19

Martes 21/12/10 (1 hora)

19.1 Ejercicios sobre Teorema de Rice, conjuntos recursivos y recursivamente enumerables

Sea W^+ el conjunto de programas escritos en el lenguaje *while*, tales que puedan recibir otros programas como input.

Ejercicio 19.1 *Sean P y Q dos programas en W^+; ¿es decidible la siguiente afirmación $[\![P]\!] = [\![Q]\!]$?*

Solución. $[\![P]\!]$ es la *semántica denotacional* de P, es decir, su significado matemático. Preguntarse $[\![P]\!] = [\![Q]\!]$ es equivalente a preguntarse si P y Q producen el mismo resultado para todo input. Sea P_i el programa cuyo índice es i, según cierta codificación (hicimos lo mismo con la Máquinas de Turing). Dado P_j, definimos el conjunto

$$\Pi_j = \{i \mid [\![P_i]\!] = [\![P_j]\!]\}$$

y vemos si se complen las condiciones del teorema de Rice. De hecho, $\Pi_j \neq \emptyset$ porque $j \in \Pi_j$; tampoco el complemento de Π_j es vació, porque no todos los programas calculan lo mismo; finalmente, Π_j es un conjunto de índices porque cualquier índice de programa equivalente está en el conjunto. Entonces se cumplen las premisas y se demuestra que Π_j no es recursivo. Por lo tanto, dados $P = P'_i$ y $Q = P''_i$ por ciertos índices i' e i'', no se puede decidir si i'' pertenece a Π'_i, es decir, si los dos programas son semánticamente equivalentes. □

Ejercicio 19.2 *Demostrar que el conjunto $\Pi = \{i \mid \varphi_i$ tiene dominio infinito$\}$ no es recursivo.*

Solución. Averiguamos si Π satisface las condiciones del teorema de Rice:

- no es vació porque por ejemplo $\lambda x.x$ tiene dominio infinito;

- no es $\mathbb{N}$ porque la función siempre indefinida no le pertenece; y
- es un conjunto de índices porque dado $i \in \Pi$ y $\varphi_j = \varphi_i$, φ_i tiene dominio infinito por hipótesis, pero φ_j también lo tiene porque son la misma función, por lo que $j \in \Pi$.

Entonces Π no es recursivo por el teorema de Rice. □

Ejercicio 19.3 *Demostrar que el conjunto* $\Pi = \{i \mid \varphi_i$ *es definitivamente* $\geq 3\}$ *no es recursivo. Que una función* f *satisfaga una condición* c definitivamente *significa que existe un* n_0 *tal que* c *se cumple para todo* $f(n)$ *con* $n \geq n_0$.

Solución. Este ejercicio es parecido al anterior. Las condiciones del teorema de Rice están satisfechas porque

- Π no es vació porque por ejemplo $\lambda x.5$ le pertenece;
- no es $\mathbb{N}$ porque $\lambda x.0$ no le pertenece; y
- es un conjunto de índices porque dado $i \in \Pi$ y $\varphi_j = \varphi_i$, φ_i es definitivamente ≥ 3 por hipótesis, pero φ_j también lo es porque son la misma función, por lo que $j \in \Pi$.

□

En general, estos conjuntos son conjuntos de índices porque definen propiedades meramente matemáticas, que no se refieren al índice de las funciones calculables.

Ejercicio 19.4 *Estudiar, al variar de* $A \subseteq \mathbb{N}$, *la recursividad del conjunto* $\Pi_A = \{i \mid \forall x(\varphi_i(x)\downarrow \Leftrightarrow x \in A)\}$.

Solución. Se ve claramente que Π es un conjunto de índices. Y también, para todo A, $\Pi_A \neq \mathbb{N}$ porque no todas las funciones tienen el mismo dominio. Entonces hay que ver si $\Pi_A = \emptyset$ por algún A, es decir, si existen conjuntos de números naturales que no son dominio de ninguna φ_i. Pero el dominio de una función calculable parcial es un conjunto RE, por lo que los A cuyo Π es recursivo (por ser vacío) son todos y sólo los conjuntos que *no* son recursivamente enumerables. □

Ejercicio 19.5 *Estudiar la recursividad de* $\Pi = \{i \mid \exists j(\varphi_i(j) = \varphi_j(i))\}$.

Solución. Π es recursivo porque es exactamente $\mathbb{N}$. De hecho, para todo i existe un j tal que $\varphi_i(j) = \varphi_j(i)$: es el mismo i. □

Capítulo 20

Martes 11/01/11 (1 hora)

20.1 Ejercicios sobre conjuntos recursivos y recursivamente enumerables

Estos dos lemas nos dan más herramientas para estudiar la recursividad de conjuntos (o lenguajes, que es lo mismo).

Lema 20.1 *Si A es el codominio de una función $f : \mathbb{N} \mapsto \mathbb{N}$ calculable total y monótonamente creciente, entonces A es recursivo.*

Demostración. Dado n, se puede decidir si $n \in A$ calculando f para el input 0, luego para 1, 2, etc. Siendo f monótonamente creciente, existe un i tal que $f(i) \geq n$, y hay dos casos:

- $f(i) = n$; en este caso n pertenece al codominio de f, por lo que $n \in A$ siendo A el codominio de f;
- $f(i) > n$; en este caso no se ha encontrado hasta el momento un input $j < i$ por el que $f(j) = n$ y, siendo f monótonamente creciente, tal input no se va a encontrar nunca; entonces se puede decir que n no pertenece al codominio de f y tampoco pertenece a A.

Siendo f calculable total, $f(i)$ se calcula en un tiempo finito, por lo que la pertenencia de n a A es decidible y A es recursivo. □

Lema 20.2 *Si A es finito entonces es recursivo.*

Demostración. Es fácil encontrar un procedimiento (en forma de MdT o de otro formalismo) M que compare su input x con todos y cada uno de los elementos de A. Si uno de estos elementos es x entonces M contesta que $x \in A$; si no es así, contesta que $x \notin A$ después de haberlo comparado con todo el conjunto. □

Notamos que si un conjunto $A \neq \emptyset$ es finito entonces no puede ser un conjunto de índices, porque cada índice $i \in A$ es acompañado por un conjunto enumerable de índice de funciones iguales.

Ejercicio 20.1 *Estudiar la recursividad de* $\Pi = \{i \mid \exists n \exists p \text{ primo } (i = p^n)\}$.

Solución. Dado el input i:

- si $i = 0$ entonces $i \notin \Pi$ (suponiendo que 0 no sea un número primo);
- si $i = 1$ entonces $i \in \Pi$ (porque $1 = p^0$ para todo p);
- si $i > 1$, todas las funciones f_p tales que $p \geq 2$ y $f_p(n) = p^n$ son
  - calculables;
  - totales;
  - monótonamente crecientes

  entonces (por el lema anterior) es decidible si i pertenece al codominio de una f_p, es decir, si existe n tal que $i = p^n$. Ahora basta con considerar todos los números primos hasta i y su f_p correspondiente:
  - si i pertenece al dominio de f_p entonces la respuesta es afirmativa;
  - si no pertenece, entonces se considera el siguiente número primo (notamos que establecer si un número es primo es un problema decidible);
  - si no pertenece a ningún codominio, entonces la respuesta es negativa.

  Como hay que considerar un número finito de número primos, la pertenencia a Π es decidibile y Π es recursivo.

□

Capítulo 21

Jueves 13/01/11 (2 horas)

21.1 Un pequeño resumen

En las últimas clases hemos estado viendo algunas herramientas para demostrar

- que un problema es decidible
  - se demuestra, por ejemplo, proporcionando una MdT que lo resuelva
  - también se puede transformar el problema en la pertenencia a un conjunto y demostrar que es recursivo
- que un conjunto (lenguaje) es recursivo
  - el codominio de una función calculable total monótonamente creciente es recursivo
  - un conjunto finito es recursivo
  - se puede proporcionar una MdT que decida la pertenencia
- que un problema no es decidible
  - se puede transformar el problema en la pertenencia a un conjunto y demostrar que no es recursivo
  - o dar alguna prueba "intuitiva" de que un algoritmo para resolver el problema no puede existir
- que un conjunto (lenguaje) no es recursivo
  - reduciendo a él un conjunto del que ya se sabe que no es recursivo
  - utilizando el teorema de Rice
  - demostrado que no existe una MdT que decida la pertenencia

Ejercicio 21.1 *Sea C el codominio de la función*

$$\psi = \begin{cases} \psi(0) = n_0 \\ \psi(n+1) = \varphi_n(n) * \psi(n) \end{cases}$$

Estudiar la recursividad de C.

Notamos que la recursividad de C no implica que lo podamos calcular tan fácilmente: hace falta saber cuál de las computaciones $\varphi_n(n)$ es la primera que no termina.

Ejercicio 21.2 *Sea la relación $\approx_w$ definida como*

$$\varphi_i \approx_w \varphi_j \;\Leftrightarrow\; \forall x(\varphi_i(x)\downarrow \wedge \varphi_j\downarrow \Rightarrow \varphi_i(x) = \varphi_j(x))$$

Esta relación se llama a menudo equivalencia débil. *Estudiar la recursividad de*

$$\Pi = \{k_i \mid k_i = \mu y.\varphi_i \approx_w \varphi_y\}$$

Ejercicio 21.3 *Sea C el codominio de la función*

$$\psi = \begin{cases} \psi(0) = n_0 \\ \psi(n+1) = 2\psi(n) \end{cases}$$

Estudiar la recursividad de C al variar de n_0.

Ejercicio 21.4 *Sea C el codominio de la función*

$$\psi = \begin{cases} \psi(0) = n_0 \\ \psi(n+1) = h(n)\psi(n) \end{cases}$$

Estudiar la recursividad de C al variar de n_0 y de la función calculable total h.

Capítulo 22

Jueves 20/01/11 (2 horas)

22.1 Algunos ejercicios sin resolver

Ejercicio 22.1 *Sea*

$$\psi(i,j) = \begin{cases} 1 & \text{si } \varphi_i(k)\downarrow \text{ para algún } k < j \\ \perp & \end{cases}$$

¿Es calculable? ¿Es total?

Ejercicio 22.2 *Sea*

$$\psi(i,j) = \begin{cases} 1 & \text{si } \varphi_i(k)\downarrow \text{ para algún } k > j \\ \perp & \end{cases}$$

¿Es calculable? ¿Es total?

Ejercicio 22.3 *Sea*

$$\psi(x) = \begin{cases} \varphi_x(x)+1 & \text{si } \varphi_x(x)\downarrow \\ \perp & \end{cases}$$

- *demostrar que es calculable*
- *mostrar que ψ no se puede extender a una función calculable total.*

Ejercicio 22.4 *Sean α y β dos funciones calculables tales que $dom(\alpha) = A$ y $dom(\beta) = B$ con $A \cap B = \emptyset$.*

- *¿existe necesariamente una función calculable γ tal que $\{\gamma(x)|x \in A\} = \{0\}$ y $\{\gamma(x)|x \in B\} = \{1\}$?*
- *¿existe necesariamente una función calculable **total** γ tal que $\{\gamma(x)|x \in A\} = \{0\}$ y $\{\gamma(x)|x \in B\} = \{1\}$?*

Ejercicio 22.5 (sobre el Teorema $s-m-n$) *Mostrar que existe una función calculable total $g : \mathbb{N} \mapsto \mathbb{N}$ tal que*

$$\varphi_{g(i,j)}(x) = \varphi_i(x) + \varphi_j(x)$$

donde la suma es definida sólo si $\varphi_i(x)$ y $\varphi_j(x)$ son las dos definidas.

Ejercicio 22.6 (sobre el Teorema $s-m-n$) *Construir una función calculable total f tal que $f(m,n)$ es un índice de φ_m compuesta consigo misma n veces (es decir, $\varphi_m \circ \cdots \circ \varphi_m$).*

Ejercicio 22.7

- *demostrar la existencia de una función calculable total g tal que*

$$\varphi_{g(i)}(j) = \begin{cases} \varphi_j(j)+1 & \text{si } \varphi_i(i)\downarrow \\ \perp \end{cases}$$

- *demostrar que, para todo i, se puede extender $\varphi_{g(i)}$ a una función total*
- *usar la g del primer punto para demostrar que es indecidible si una función calculable se puede extender a una función calculable total.*

Ejercicio 22.8 *Demostrar que la función total f tal que $f(n)$ es el índice más pequeño por el que $\varphi_n = \varphi_{f(n)}$ no es calculable.*

Ejercicio 22.9 *Sea f una función calculable total, y $x \in \mathbb{N}$; demostrar que el conjunto $f^{-1}(dom(\varphi_x)) = \{y \mid f(y) \in dom(\varphi_x)\}$ es recursivamente enumerable.*

Capítulo 23

Martes 25/01/11 (1 hora)

23.1 Más ejercicios

Ejercicio 23.1 *Demostrar que la cardinalidad de un conjunto A es menor que la de los números naturales si y sólo si todo $B \subseteq A$ es recursivo.*

Solución. Una dirección es fácil: si A tiene menos elementos que $\mathbb{N}$, entonces es finito. Cada subconjunto B de A será finito y, por lo tanto, recursivo.

Además, supongamos que todo $B \subseteq A$ sea recursivo. Entonces, si A tuviera un número infinito de elementos, existiría una correspondencia biunívoca entre A y $\mathbb{N}$ (el caso que A sea más grande que $\mathbb{N}$ se reduce al anterior). Como todo $B \subseteq A$ es recursivo, la biyección asegura que también todo $C \subseteq \mathbb{N}$ es recursivo, pero sabemos que esto no es cierto. Por lo tanto, suponer que A es infinito genera una contradicción, por lo que se concluye que A es finito. □

Ejercicio 23.2 *Sean f y g funciones recursivas totales. Sea g inyectiva, y sea su condominio A_g recursivo. Entonces vale lo siguiente:*

$$\forall x \in \mathbb{N}.\ f(x) \geq g(x) \quad \Rightarrow \quad \textit{el codominio de } g \textit{ es recursivo}$$

Capítulo 24

Jueves 24/02/11 (2 horas)

24.1 La computabilidad, en concreto

- ¿es realmente importante lo que estamos estudiando?
- es decir, desde siempre los programas se hacen, si usan, y más o menos funcionan; ¿de qué sirve todo esto?
- con los avances de la tecnología, ¿va a cambiar algo en la forma de mirar a los problemas insolubles?

Una observación:

- puede que a alguien la teoría de la computabilidad no le diga nada: efectivamente, hay mucha gente que opina de esta forma; es decir, les da igual que "no se pueda" una cosa u otra, si al final consiguen implementar el program que quieren y lo pueden vender
- en cierto sentido, es como hablar de resultados matemáticos teóricos a alguien que no está interesado
- ¿cómo puedo vender un *Ribera del Duero Crianza 2005* de Bodegas Resalte Peñafiel a una persona que lo que quiere es emborracharse?
- y vosotros, ¿quereis el vino bueno o sólo emborracharos?
- pero hacia donde vamos es demostrar que la computabilidad no es cosa para los entendidos, sino que tiene un impacto en el día a día del informático.

Ejemplo: el problema de especificar los *test cases*.

Para los que hasta dudan de que la computabilidad sea importante *desde el punto de vista matemático*:

- ¿y si tuvieramos un programa halts que, dado un programa P y un input w, decide si $P(w)$ termina o no mirando a *todos* los estados de P hasta que encuentra un estado *exacta* y *totalmente* repetido?

Y para los aficionados de ciencia ficción:

- ¿qué puede cambiar con los nuevos paradigmas de computación (to be continued...)?

24.2 Tareas indecidibles en el día a día de un informático: verificación y análisis estático

Podemos definir informalmente un *analizador* como un programa que trabaja con otros programas, *sin ejecutarlos*. ¿Por qué *sin ejecutarlos*? Porque deberíamos considerar un número infinito (o, en todo caso, demasiado grande) de situaciones donde el programa se pueda ejecutar.

En el ejemplo del test case, podemos ver al menos tres dificultades (el ejemplo se concentra sobre todo en la segunda):

- saber lo que se le va pedir al programa (*especificación*)
  - la definición de los tipos de triángulos, los inputs que no hay de considerar como válidos
- individuar un número suficiente de test cases para que nos podamos convencer de que el programa cumple con su tarea
  - un input para cada caso significativo de ejecución
- esperar que el programa no tenga fallos que ni siquiera una buena batería de tests puede descubrir
  - por ejemplo que, si el program responde bien para $(5, 5, 5)$, lo normal es que también responda bien para $(10, 10, 10)$

El tercer punto nos ofrece otra perspectiva:

- el hipótesis que el programa no tenga fallos ocultos puede ser razonable si somos nosostros los autores del código y nuestra tarea es comprobar que sea correcto;
- sin embargo, si el autor del código es otra persona y *no nos podemos fiar de él*, entonces podría haber metido *a posta* en el programa algo por el que el mismo programa responde bien para $(5, 5, 5)$ y hasta $(9, 9, 9)$, pero no para $(10, 10, 10)$

24.3 Ejemplos de analizadores

- un simple *duplicador* o copiador de programas
- un componente de un compilador: un analizador lexical
- otro componente de un compilador: un analizador sintáctico
- un algoritmo de *type checking* o *type inference*
- un compilador completo
- un verificador de propiedades

Las tareas que aparecen em el listado tienen diversos niveles de dificultad, pero en todo caso podemos individuar

- una tarea a ejecutar, con su dificultad
- un resultado
- lo que se pide a un analizador: si siempre tiene que sacar el resultado (correcto)

Ejemplo 24.1 (Duplicador de programas)
- *tarea: copiar el código; es una tarea decidible y fácil*
- *resultado: el código duplicado*
- *siempre lo puede hacer, y siempre lo tiene que hacer*

Ejemplo 24.2 (Analizador lexical/sintáctico)
- *tarea: ver si en el código hay errores de sintáxis; también es una tarea decidible si la sintaxis está bien definida*
- *resultado: "sí" o "no", dependiendo de si hay errores en el código (también decir cuáles)*
- *siempre lo puede hacer, y siempre lo tiene que hacer*

Sin embargo hay tareas más difíciles, hasta ser indecidibles.

Ejemplo 24.3 (Algoritmo de type checking/inference)
- *tarea: garantizar que en ninguna ejecución hay un valor de cierto tipo donde no era esperado (por ejemplo, en un lenguaje funcional, una función en lugar de un número)*
- *resultado: "sí" o "no"*
- *el algoritmo tiene que terminar siempre y ser razonablemente eficiente (de otra forma no sería muy útil); entonces*

- *tendrá que decir que "no" para todo programa cuyo comportamiento es incorrecto*
- *pero también dirá que "no" para programas que en realidad no generan errores de tipo*

Ejemplo 24.4 (Problemas con el type checking/inference) *En este programa la condición es verdadera siempre, pero ningún analizador es capaz de verlo; por lo tanto, ya que el type system cree que un error de tipo es posible, lo normal es que no acepte el programa.*

```
1  if <complex test> then 42 else <type error>
```

Hay también casos más sutiles (por ejemplo en Haskell, cuyo sistema de tipos es muy exigente) donde se rechazan programas que no generan errores de tipo.

Capítulo 25

Jueves 03/03/11 (2 horas)

Observación preliminar: en la terminología usada hasta ahora, un analizador es un algoritmo que, dado un programa, puede

- (*analizadores aceptores*) contestar *sí* o *no* (aceptar o no aceptar); en este caso, el objetivo es *decidir un lenguaje* (no necesariamente un lenguaje de programación: en general, un conjunto de cadenas de símbolos)
  - por ejemplo, un analizador sintáctico para Java decide el mismísimo lenguaje *Java* (que no tiene por qué ser el mismo lenguaje de programación en el que el analizador es implementado); está claro que Java como lenguaje es decidibile
  - pero normalmente, para problemas indecidibles de semántica, el analizador decide *una aproximación* (desde abajo) del lenguaje considerado (por ejemplo, lo programas seguros según cierta propiedad de seguridad)
  - ¿por qué no es correcto decir que semidecide el lenguaje de interés?
- (*analizadores calculadores*) proporcionar un output que describa algunas propiedades de los programas que se le pueden dar como input: al fin y al cabo se trata de calcular un función

25.1 Manos a la obra

Hasta ahora hemos visto ejemplos como

```
1 if <complex test> then 42 else <type error>
```

donde se supone la existencia de un "complex test" o algo que un analizador no puede tratar. Pero necesitamos ver *en vivo y en directo* programas para los que los analizadores fallan.

El analizador Interproc[1] nos permite acercarnos a estos problemas. Pero antes tenemos que estudiar el marco teórico para formalizar los análisis.

25.2 Data Flow Analysis

- Transparencias dataFlowAnalysis.pdf (1-43, con huecos; 87-97; 103; 113)

Ideas fundamentales:

- Control Flow Graph con bloques elementales
- el uso de las ecuaciones
- liveness analysis se puede ver como una abstracción (véase abajo), siendo la propiedad de liveness indecidible (¿dónde se pierde información?)
- un análisis hacia delante es una ejecución abstracta del programa, donde se ha cambiado la semántica

[1] `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`

Capítulo 26

Viernes 04/03/11 (2 horas)

26.1 Interpretación Abstracta en 2 horas

En esta discusión vamos a considerar como ejemplo un type checker: es un analizador aceptor que sólo acepta un programa cuando está seguro de que no se van a generar errores de tipo durante la ejecución.

Semántica concreta Hablamos de lenguajes imperativos sencillos con variables globales, sin objetos, etc. Lo primero es definir la *semántica* (que llamaremos *concreta* para distinguirla de la *abstracta*).

Un *estado* es una función que asigna a cada variable un valor. Dado un estado de input, nos interesa el estado que se obtiene después de cierto número de pasos (y sobre todo al final de la computación). El estado también sabe el punto de programa en el que la computación se encuentra.

- ejemplos (deberían ser conceptos conocidos)

Semántica collecting La semántica *collecting* define, *para cada punto de programa*, el conjunto de estados posibles. Por ejemplo, dado el programa

```
1 begin
2   x = 0;
3   while (x<100) do
4     x = x+2;
5   done;
6 end
```

el valor de la variable x nada más entrar en el loop está en el conjunto $\{0, 2, .., 98\}$. Lo podemos ver mediante Data Flow Analysis, que nos proporciona un método y también nos hace entender que el problema es indecidible.

Ya de por sí, la semántica collecting es una *abstracción* de la otra, es decir, pierde información.

Ejemplo 26.1 *Consideremos el siguiente programa:*

```
1  begin
2    x = x+2;
3  end
```

donde se supone que el valor inicial de x *es el input.*

Entonces la semántica concreta dice que para todo estado de input σ tal que $\sigma(x) = n$, el estado de output correspondiente (hay un σ' para cada σ) σ' satisface $\sigma'(x) = n+2$. En cambio la semántica collecting puede hablar sólo con conjuntos de estado en cada punte de programa, por lo que

- *el conjunto de estados de input (linea 0 del programa) es $\{\sigma|\sigma(x) = n, n \in \mathbb{Z}\}$, y*
- *el conjunto de estados de output (linea 3 del programa) es el mismo: $\{\sigma|\sigma(x) = n, n \in \mathbb{Z}\}$.*

Esto quiere decir que todo valor para x *es posible antes y después de la ejecución, pero no puedo expresar con la semántica collecting la dipendencia input-output (que la variable aumenta su valor de 2).*

Aunque la semántica collecting es una abstraccíon (es decir, algo menos preciso) de la semántica concreta, calcularla sigue siendo algo indecidible. Por lo tanto, necesitamos una abstracción de ella.

La aproximación En nuestro planteamiento, el analizador que queremos estudiar está interesado en sacar del programa información acerca de los puntos del programa. Como ejemplo, podemos pensar en un algoritmo de type inference que para cada punto de programa quiere calcular el tipo de cada variable, asumiendo (normalmente no es el caso, a no ser que sean de hecho variables distintas con el mismo nombre) que las variables puedan cambiar de tipo. En cierto sentido, el analizador quiere calcular la semántica collecting, pero, dada su indecidibilidad, se tiene que conformar con una aproximación.

Es decir, la información obtenida por el analizador en el punto de programa i *incluye* conceptualmente todos los valores de la semántica collecting en i (y posiblemente algunos o infinitos más). ¿Cómo representa el analizador esta información? Con una formalización que depende de su capacidad de calcular; en el caso de los tipos en lenguajes funcionales, el sistema de tipos que se usa determina la representación de dicha información.

Ejemplo 26.2 *Un sistema de tipos polimórfico puede expresar más tipos que uno monomórfico, así que, en general, se ajusta más a la semántica collecting. En cambio, un sistema di tipos que sólo posea* **int** *fallará cada vez que una variable tiene un tipo funcional.*

Vamos a concretar. Sea $\mathcal{V}$ el conjunto de valores que las variables pueden tener. Entonces el *dominio concreto* es $\mathcal{C} = \wp(\mathcal{V})$, y la semántica collecting asigna a cada punto de programa en elemento del dominio concreto.

La abstraccíon de la semántica collecting se obtiene definiendo un *upper closure operator* (uco) ρ, que es una función ed $\mathcal{C}$ a $\mathcal{C}$ con tres propiedades:

- monotona: $v_1 \subseteq v_2$ implica $\rho(v_1) \subseteq \rho(v_2)$
- idempotente: $\rho(\rho(v)) = \rho(v)$
- extensiva: $\rho(v) \supseteq v$

ρ identifica el *dominio abstracto*. Más formalmente, el dominio abstracto es el conjunto de los puntos fijos de ρ, es decir, los v tales que $\rho(v) = v$. Está claro que $\rho(\mathcal{V}) = \mathcal{V}$ (normalmente representado con $\top \approx$ *top*). Y, normalmente, $\rho(\emptyset) = \emptyset$ (normalmente representado con $\bot \approx$ *bottom*).

Ejemplo 26.3 *Coches con marcas de coches.*

Ejemplo 26.4 *Personas con nacionalidades (de región, país o continente: dominio más estructurado).*

Ejemplo 26.5 *Artículos científicos con keywords: más complicado porque a cada artículo le corresponden diversas keywords.*

Ejemplo 26.6 *Sea $\mathcal{V}$ el conjunto de valores de un lenguaje funcional. Entonces la inferencia de tipos corresponde a un upper closure operator porque*

- *cada tipo es la representación de un conjunto de valores*
- *también hay un tipo más general que corresponde al mismo $\mathcal{V}$ y básicamente quiere decir que no conocemos nada*
- *dado $v \in \mathcal{V}$, el conjunto $\rho(v)$ (o el tipo que le representa) viene a ser el conjunto más pequeño (que corresponde a un tipo) que incluye todos los valores de v*
  - *si $v = \{2, 4\}$ entonces el* valor abstracto *$\rho(v)$ es $\mathbb{Z} \approx$* **int**
  - *si $v = \{\lambda x.x, \lambda x.1\}$ entonces $\rho(v) \approx$* **int**$->$**int**
  - *si $v = \{1, 2, 3, \lambda x.x\}$ entonces, en general, no hay un tipo que le corresponde y ρ devuelve $\top = \mathcal{V}$*

El requisito fundamental del dominio abstracto y, globalmente, del analizador es que *no se pierda nada* de la semántica collecting (*correctness* o *soundness*). Es decir, si la collecting dice que en cierto punto de programa x puede ser cualquier número entre 10 y 100, entonces el analizador tiene que calcular para x algo que incluya todos los números entre 10 y 100.

Vamos concretando: en nuestros ejemplos $\mathcal{V}$ es $\mathbb{Z}$ (sólo números enteros). Hay varias abstracciones que podemos considerar:

- parity
- zeroness

- sign
- intervals
- congruences

Y también algunas abstracciones *relacionales* que pueden decir algo de los valores de las variables en su conjunto:

- octagons
- polyhedra

Ejercicio 26.1 *Para todas estas abstracciones, define el ρ correspondiente.*

Un analizador intenta aproximar la semántica collecting por medio de una abstracción. En general hay una pérdida de precisión, debido a

- (trivial) la propiedad que se modela con ρ; por ejemplo, en sistema de tipos, decir que una variable en **int** no dice mucho de su valor; o, en el caso de los intervalos, decir $x \in \{3, 5, 8\}$ es más preciso que decir $x \in [3, 8]$;
- (menos trivial) el hecho que no siempre es posible sacar la mejor aproximación posible de un valor: esto está relacionado con la transfer function (y también con cómo se evaluan las expresiones) y se llama *incompleteness*; se ve a la hora de propagar los valores abstractos en el código del programa.

Capítulo 27

Jueves 10/03/11 (2 horas)

27.1 El analizador Interproc

Inteproc es un analizador calculador porque para todo programa correcto proporciona una representación de algunas de sus propiedades. También incluye un analizador aceptor que verifica la sintaxis del input y rechaza los programas incorrectos, pero este analizador no nos interesa (entre otras cosas, porque el problema es decidible).

Interproc puede usar varios dominios abstractos; para cada dominio, implementa la semántica abstracta correspondiente. El resultado es una descripción abstracta (*invariante*) de los valores de las variables en cada punto de programa. Esto quiere decir que *en toda ejecución* que pase por ese punto de programa los valores concretos se pueden representar por medio del valor abstracto.

Nos concentramos en programas que siempre calculan 0 como valor final de la variable `a`. Queremos que Interproc nos diga (o nos ayude a decir) si cierto programa calcula efectivamente esta función, que llamamos φ_0. Es decir, Interproc intenta averiguar una propiedad que es semántica. Por el teorema de Rice, sabemos que no existe un analizador que decida esta propiedad, y vamos a ver cómo esto sucede en la práctica.

Ejemplo 27.1 *En el primer ejemplo, trivial, el dominio abstracto de los intervalos (Box) permite a Interproc decir que el programa sí calcula* φ_0.

```
1  var a:int;
2  begin
3    a = 0;
4  end
```

Notamos que el valor inicial de `a` *se considera dado como input, pero en este caso no importa porque lo primero que hace el programa es asignarle* 0.

Ejemplo 27.2 *Este ejemplo no se puede decidir con Box, pero sí se puede decidir con Octagon. El problema no es el resto, sino el hecho de que no se conoce el valor inicial de* `a`.

```
1  var a:int;
2  begin
3    a = a-a;
4  end
```

Ejemplo 27.3 *De hecho, en este ejemplo el valor inicial de* a *es conocido y Box puede inferir el resultado correcto.*

```
1  var a:int;
2  begin
3    a = 7;
4    a = a-a;
5  end
```

Ejemplo 27.4 *Y también en este ejemplo.*

```
1  var a:int, b:int, c:int;
2  begin
3    b = 2;
4    c = 2*b-b+1;
5    if (c>b) then
6      a = 0;
7    else
8      a = 3;
9    endif;
10 end
```

Ejemplo 27.5 *Pero este ejemplo es demasiado difícil para Box, incluso si ignora totalmente el input. El motivo es que el procedimiento* MC *es un función cuyo análisis es difícil (de hecho, en su día McCarthy, el inventor del lenguaje Lisp, la propuso como* test case *para la verificación formal de programas). De todas formas, Octagon es capaz de inferir que el programa sigue calculando φ_0, porque sabe tratar la asignación* c=b+1 *y la siguiente condición* c>b.

```
1  proc MC(n:int) returns (r:int)
2  var t1:int, t2:int;
3  begin
4    if (n>100) then
5      r = n-10;
6    else
7      t1 = n + 11;
8      t2 = MC(t1);
9      r = MC(t2);
10   endif;
11 end
12
13 var a:int, b:int, c:int;
14 begin
15   a = 56;
```

```
16      b = MC(a);
17      c = b+1;
18      if (c>b) then
19        a = 0;
20      else
21        a = 3;
22      endif;
23    end
```

Ejemplo 27.6 *Este ejemplo es parecido al anterior, pero todavía más difícil, hasta el punto que Octagon no es capaz de resolverlo. Sin embargo, se puede resolver con Polyhedra (Polka)*

```
1   proc MC(n:int) returns (r:int)
2   var t1:int, t2:int;
3   begin
4     if (n>100) then
5       r = n-10;
6     else
7       t1 = n + 11;
8       t2 = MC(t1);
9       r = MC(t2);
10    endif;
11  end
12
13  var a:int, b:int, c:int;
14  begin
15    a = 56;
16    b = MC(a);
17    c = 2*b-100;
18    if (c/2<b) then
19      a = 0;
20    else
21      a = 3;
22    endif;
23  end
```

Ejemplo 27.7 *Finalmente, el último ejemplo es demasiado difícil incluso para Polyhedra.*

```
1   proc MC(n:int) returns (r:int)
2   var t1:int, t2:int;
3   begin
4     if (n>100) then
5       r = n-10;
6     else
7       t1 = n + 11;
8       t2 = MC(t1);
9       r = MC(t2);
10    endif;
```

```
11  end
12
13  var a:int, b:int, c:int;
14  begin
15    a = 56;
16    b = MC(a);
17    c = 2*b-100;
18    if (c<=b) then
19      a = 0;
20    else
21      a = 3;
22    endif;
23  end
```

¿Y si Interproc usara dominios abstractos más complejos y/o técnicas más refinadas? El teorema de Rice nos dice que siempre encontraremos un programa que es demasiado complicado para el análisis. Además, la eficiencia es una punto fundamental para un analizador que se quiera usar en la práctica, así que no siempre podemos usar el mejor dominio abstracto "posible" (es decir, que seríamos capaces de implementar).

Interproc incluye otras posibilidades que lo hacen más potente, pero esto no cambia el cuadro general.

Capítulo 28

Viernes 11/03/11 (2 horas)

28.1 Análisis estático del heap

En lenguajes con punteros, la memoria durante la ejecución se representa como un *heap*. La posibilidad de alocar dinámicamente locaciones de memoria complica mucho el comportamiento de los programas y la tarea de un analizador.

Entre la propiedades que puede ser interesante estudiar en este entorno hallamos:

- nullity
- sharing
- points-to, aliasing
- reachability
- cyclicity
- shape
- termination
- cost (number of instructions)
- memory usage

Capítulo 29

Jueves 24/03/11 (2 horas)

29.1 Code Protection

Transparencias codeProtection.pdf (de 3 a 23).

29.2 Code Obfuscation

Transparecias codeProtection.pdf (de 24 a 45).

29.3 The Halting Problem for Reverse Engineers

De Internet: **http://indefinitestudies.org/2010/12/19/the-halting-problem-for-reverse-engineers/**

I often have the feeling that technically savvy people don't have a very high opinion of academia, and this is particularly true of security people. They have to deal with low-level details such as hardware architecture, operating systems internals (more or less documented), proprietary protocols and data structures, all of which require very specialized knowledge. Logic, theorems and algorithms don't have a predominant place in that picture.

For the past few years I have been working on these subjects, and found some fundamental theorems to be actually useful in understanding the security properties of computer architectures. One of them is, of course, the undecidability of the halting problem. In essence, it says that we can not know if the computation of a program on some input will ever terminate. So what? Who cares if a program terminates, what we want is to find vulnerabilities and unpack malware samples, right?

The importance of the undecidability of the halting problem lies in its generality. In particular we can see Rice's theorem as a generalization of this result, and to put it very simply, it says that whatever properties of programs you're

interested in, no program can tell if this property holds for every program (i.e. it is undecidable).

This is very bad news for all of us, since basically everything about programs is undecidable. Say that you are interested in finding functions that do out-of-bounds memory writes (or as I said, any other property), Rice's theorem says that there is no program that will give you a correct answer all the time. You must accept that your program sometimes fails or infinitely loops.

I want to emphasize how bad this is. Do not let the terminology used in the theorems confuse you. In particular, the notions of input, output, and function computed by a program do not map nicely to binaries. An output is anything that gets modified by your program - any register or memory location, as soon as it is touched by an instruction, is an output. And basically, everything about outputs is undecidable. As a consequence, simple tasks such as disassembling are undecidable.

For instance, take this seemingly innocent indirect jump:

```
jmp [eax]
```

If eax is an output of instructions before it, no luck, its value is undecidable. You can run the program, write the value down, and assume it will not change, but you have no guarantee that it will not change at a given date. Undecidable. You could argue that eax can only take a finite number of values, and hence disassembling is still possible, just very intractable. But that would be without counting on self-modifying code. SMC (think packers) is the scourge of disassemblers because it gives the ability to transfer control to an output. Since I can't decide the value of the output, I can't disassemble.

To sum things up, here are a few direct consequences of the undecidability of the halting problem:

1. you can't decide the target of indirect jumps, reads and writes
2. you can not decide if a particular memory address is code, data, or both
3. 3. you can't decide values written in memory
4. 4. you can't decide the numbers of occurrences of loops
5. 5. you can't decide if control flow can reach a given instruction
6. 6. whatever you see in a given run can change arbitrarily in another run
7. 7. disassembling is undecidable
8. 8. unpacking is undecidable

I will leave how all this led to the bad habit of relying on "heuristics" to a further post. Stay classy!

Capítulo 30

Viernes 25/03/11 (2 horas)

Code Obfuscation: transparecias codeObfuscationAttack.pdf (de 3 a 25 y de 50 a 74) y codeObfuscationDefense.pdf (todas).

Capítulo 31

Jueves 31/03/11 (2 horas)

31.1 Software Watermarking

Transparencias codeProtection.pdf (de 46 a 59) y softwareWatermarking.pdf (de 1 a 23).

Capítulo 32

Viernes 1/04/11 (2 horas)

32.1 Software Watermarking

Transparencias softwareWatermarking.pdf (de 24 a 52).

Capítulo 33

Jueves 7/04/11 (2 horas)

33.1 Complejidad

33.2 Introducción

- requisitos para una teoría de la complejidad
- el ejemplo de las Random Access Machines
- complejidad en espacio y en tiempo
- clases de complejidad conocidas y relaciones entre ellas

Capítulo 34

Viernes 8/04/11 (2 horas)

34.1 Complejidad y MdTs con k cintas

- descripción
- teorema sobre la relación entre MdTs con una cinta y con k cintas
- teorema de acceleración lineal

34.2 Notaciones para la complejidad

$\mathcal{O}(n)$, $\Omega(n)$, $\Theta(n)$.

Capítulo 35

Jueves 28/04/11 (2 horas)

35.1 Método de evaluación

- la mayoría ya tenemos el examen de la primera parte hecho
- el resto de la nota es la práctica de la que vamos a hablar hoy
- sólo para subir nota puedo preparar un examen sobre la segunda parte (análisis estático, complejidad, etc.)

35.2 Prácticas

Cada tarea será parte de la evaluaciín final. Se trata de profundizar en un tema entre los propuestos y preparar una presentación en clase de 25 minutos, a finales de curso. La presentación deberá fomentar una discusión (son temas abiertos e importantes en informática, inteligencia artificial, filosofía).

- 25 minutos (más 5 de preguntas) es lo que dura normalmente una charla en un congreso; no hay problema si se quiere hacer una presentación más larga, pero es bueno acostumbrarse a tener límites de tiempo;
- aunque *no* sea una buena norma para las presentaciones, os voy a pedir unas transparencias un poco "densas" (con bastante texto) para que sean útiles en el futuro;
- en la presentación, junto con una descripción del tema, me interesa también (en la medida de lo posible) *vuestra opinión*.

35.2.1 Los argumentos "Gödelianos" de John Lucas y Roger Penrose

- el planteamiento matemático: en clase hablaremos de la conexión entre computabilidad y teoremas de Gödel

- las implicaciones en el *problema mente-máquina*
- reacciones favorables y contrarias en la comunidad científica
- suena mal, pero un buen punto de partida puede ser la página de Wikipedia en inglés sobre Penrose

35.2.2 Il problema mente-máquina, en general

- es un pelín off-topic (se acerca mucho a la Inteligencia Artificial) y muy amplio
- demostración automática: el problema de Robbins y más cosas; ajedrez
- cerebro, mente y computación
- redes neuronales; robótica

35.2.3 Super-Turing Computing

- también se conoce como *hypercomputation*
- se trata de (propuestas de) modelos de computación que van "más allá" (calculan más cosas) de las MdTs
- lo que me interesa no es una panorámica completa, sino profundizar en algunas de las propuestas (hay muchas) y empezar a ver las hipótesis que están por debajo, sus puntos fuertes y sus puntos débiles

35.2.4 Grados de indecidibilidad

- "degree of undecidability"
- se puede construir una hierarchía de problemas indecidibles: de menos "imposibles" a más "imposibles"
- bastante teórico

35.2.5 Quantum Computing

- aquí se habla de una técnica que, al menos en la teoría, se concibió hace mucho tiempo
- en algunos casos, se puede decir que se empiezan a obtener *resultados* (es decir, mejoras importantes en la complejidad de algunos algoritmos conocidos
- su posible papel en la criptografía

35.3 Complejidad

- aproximación de Stirling para el factorial:

$$\forall n > 0. \quad \left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \leq \quad n! \quad \leq \quad \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n+(1/12n)}\right)$$

35.3.1 Mergesort y su complejidad

- se genera esta *ecuación de recurrencia*:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(n/2) + \Theta(n) & n > 1 \end{cases}$$

35.3.2 El Master Theorem

$$T(n) = aT(n/b) + f(n)$$

- si $f(n) = \mathcal{O}(n^{\log_b a - \varepsilon})$ para un $\varepsilon > 0$ entonces $T(n) = \Theta(n^{\log_b a})$
- si $f(n) = \Theta O(n^{\log_b a})$ entonces $T(n) = \Theta(n^{\log_b a} \log_2 n)$
- si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ para un $\varepsilon > 0$ y $af(n/b) \leq cf(n)$ para $c < 1$ y n bastante grande, entonces $T(n) = \mathcal{O}(f(n))$

35.3.3 Quicksort y su complejidad

```
1  function quicksort(array) {
2    var list less, greater;
3    if (length(array) <= 1) {
4      return array;
5      // an array of zero or one elements is already sorted
6    }
7    select and remove a pivot value pivot from array;
8    foreach x in array {
9      if (x <= pivot) then {
10       append x to less;
11     } else {
12       append x to greater;
13     }
14   }
15   return concatenate(quicksort(less),pivot,quicksort(greater
       ))
16 }
```

35.3.4 Límite inferior para la complejidad de un algoritmo de sorting

- un array de n elementos tiene $n!$ permutaciones posibles
- cada comparación entre elementos excluye, como mucho, la mitad de las premutaciones
- se necesitará un número h de comparaciones, donde $n! \leq 2^h$
- que se escribe como $h \geq \log_2(n!)$
- por la aproximación de Stirling, $n! > \left(\frac{n}{e}\right)^n$
- $h \geq \log_2 \left(\frac{n}{e}\right)^n = n \log_2 n - n \log_e n = \Theta(n \log_2 n)$
- un algoritmo que haga al menos k comparaciones entre elementos de un array tiene como complejidad al menos k
- siendo las comparaciones (entre números) la operación fundamental en muchos algoritmos, a veces se coge su número como la medida de la complejidad en tiempo

Capítulo 36

Jueves 05/05/11 (2 horas)

36.1 Complejidad en tiempo y en espacio

36.1.1 Tiempo

Definición 36.1 *Se dice que un* problema *está en la clase* $TIME(f(n))$ *si existe un algoritmo que lo resuelve en* $\mathcal{O}(f(n))$.

Definición 36.2 *La clase de complejidad* $\mathcal{P}$ *de los problemas resolubles en* tiempo polinomial *se define como*

$$\mathcal{P} = \bigcup_{k \geq 1} TIME\left(n^k\right)$$

Esta clase de complejidad goza de una propiedad que justifica el hecho que usemos sólo órdenes de magnitud, sin considerar factores constantes: es *invariante respecto a los modelos.*

Si la complejidad de un problema se representa con un polinomio $c_1 n_k + c_2 n^{k-1} + .. + c_k$

- el teorema de aceleración permite bajar c_1 hasta cerca de 1
- las partes con orden menor de k no juegan ningún papel significativo

entonces lo que interesa es k (de aquí, $\mathcal{O}(n^k)$).

Estas observaciones son una intuición hacia lo que nos interesa: que si un problema se resuelve en tiempo polinomial con una Máquina de Turing, se resuelve en tiempo polinomial con cualquier formalismo de computación razonable. Por "razonable" entendemos

- que no de pasos demasiado estúpidos: por ejemplo, dada una lista de n elementos, calcular *en cada paso* todas las posibles permutaciones de la lista; claramente, esto en general no tiene sentido;

- que no de pasos "milagrosos": por ejemplo, dada una lista de n ciudades y las distancias entre ellas, tener una operación *elemental* dentro del lenguaje que calcule en un paso el mejor camino que toque todas las ciudades.

Se dice que la clase $\mathcal{P}$ es *cerrada* con respecto al cambio de modelo.

36.1.2 Espacio

Por otro lado, si hablamos de *complejidad espacial*, existe un teorema de *compresión lineal* que es muy parecido al de acceleración lineal y dice que se puede reducir el espacio usado por un factor constante cambiando la maquinaría (típicamente, aumentando el número de símbolos de la MdT).

En general *no* se considera como espacio de la computación el espacio ocupado por el input y por el output (si es que lo hay). De otra forma, todos los problemas serían al menos lineal en espacio, pero puede ser interesante ver que hay problemas logarítmicos. Tenemos dos clases de complejidad interesantes:

$$PSPACE = \bigcup_{k\geq 1} SPACE(n^k) \qquad LOGSPACE = \bigcup_{k\geq 1} SPACE(k\log(n))$$

También estas dos clases de complejidad son invariantes respecto a los modelos.

36.2 Modelos no deterministas

Ya sabemos que

- hay problemas que requieren explorar un espacio de soluciones para encontrar la mejor;
- se puede ver estas computaciones de forma *no determinista* como "estudia todas las soluciones al mismo tiempo y compáralas entre ellas";
- claramente, se puede ver también de manera secuencial, estudiando las soluciones una detrás de otra;
- si pudieramos estudiar todas las soluciones a la vez, la intuición nos dice que sería mucho más rápido;
- sin embargo, este modelo (por ejemplo, las MdTs no deterministas) no es hoy en día realista (con el permiso del Quantum Computing).

Intuitivamente, una MdT determinista (que, al menos en su aproximación con recursos finitos, existe) puede simular una MdT no determinista (que, hasta la fecha, no existe como modelo de computación realista) con una perdida de eficiencia exponencial.

Por eso, se consideran los problemas polinomiales deterministas $\mathcal{P}$ como *fáciles*, y los polinomiales no deterministas $\mathcal{NP}$ como *difíciles*. Está claro que

fácil y *difícil* son conceptos no formales de los que lo mejor que se puede hacer es dar una justificación informal del por qué se usan.

Una cosa interesante es que, si hablamos de espacio, el no determinismo no añade nada a nivel de problemas polinomiales: la clase *NPSPACE* es igual que *PSPACE*. Esto es coherente con la intuición que

- buscar todas las soluciones de un problema a la vez supone ahorrare mucho tiempo;
- pero buscarlas una detrás de otra no tiene por qué usar más espacio.

Una MdT no determinista decide un problema en tiempo $f(n)$ si *una de sus computaciones* no determinista termina en un estado de aceptación. No importa si muchas computaciones han terminado antes en un estado de rechazo. Este modelo de determinismo (llamado en algunos textos *angélico*) no es el único posible. Entonces la clase de problemas decidibles en tiempo no determinista polinomial es

$$\mathcal{NP} = \bigcup_{k \geq 1} NTIME(n^k)$$

que, claramente, incluye $\mathcal{P}$. Si la otra inclusión fuera también cierta se podría transformar problemas difíciles en problemas fáciles, pero esto todavía no se sabe y es uno de los desafíos más importantes de toda la informática.

Ejemplo 36.1 *El* travelling salesman problem *(TSP) es un problema muy conocido del que sólo se conocen soluciones polinomiales no deterministas o exponenciales deterministas.*

Implementar una solución polinomial (cuadrática) no determinista no es difícil:

- *en una casilla A se escribe un número muy grande, por ejemplo la suma de todas la distancias entre todas las ciudades; este número es seguramente más grande que la solución del problema;*
- *para cada computación no determinista hay dos fases:*
  - *escribir sobre la cinta una cualquiera secuencia de n números entre* 1 *y n (*$\mathcal{O}(n)$ *pasos);*
  - *averiguar que los números no se repitan (es decir, que el camino toque todas las ciudades una vez) (*$\mathcal{O}(n^2)$ *pasos porque cada vez que encuentra un número lo tiene que comprarar con los anteriores);*
  - *calcular la longitud del camino (*$\mathcal{O}(n^2)$ *pasos porque hay que buscar dentro de la matriz de las distancias); si es menor del valor de A, escribirla en A.*
- *cada computación produce una secuencia distinta y todas van* en paralelo*, así que todas habrán terminado en tiempo* n^2 *y el valor final de A será la solución.*

Otro ejemplo de computación no determinista es demostrar un problema generando todas las posibles demostraciones hasta que una de ellas sea de verdad una demostración correcta del enunciado. El proceso parará cuando dicha demostración se encuentre (puede que haya otras demostraciones).

36.3 La tesis de Cook-Karp

Se llama así porque es un poco el correspondiente de la tesis de Church-Turing (o de Church) para la complejidad. Dice algo no del todo formal y que necesita que nos entendamos sobre lo que es fácil y lo que es difícil (igual que en computabilidad había que ponerse de acuerdo sobre lo que es calculable).

Como ya hemos visto, la tesis dice que los problemas en $\mathcal{P}$ son las fáciles y los problemas en $\mathcal{NP}$ son los difíciles.

A favor de la tesis:

- $\mathcal{P}$ y $\mathcal{NP}$ son cerrado respecto al cambio de modelo (dentro de lo razonable).
- $\mathcal{P}$ resiste las reducciones $\leq_F$ son F polinomial: si tengo un problema P que se transforma en P' en tiempo polinomial, y P' está en $\mathcal{P}$, entonces P tambié está en $\mathcal{P}$.

En contra de la tesis:

- un algortimo en $\mathcal{O}(n^{100})$ es menos eficiente, en la práctica, que uno en $\mathcal{O}\left(2^{n/100}\right)$;
- existen algoritmos que requieren un tiempo exponencial en el caso peor, pero normalmente son polinomiales y además bastante eficientes.
  - programación lineal: algoritmo de Simplex, que es exponencial en el caso peor pero más eficiente que el método de las elipses que es polinomial;
  - inferencia de tipos en Standard ML: todo algoritmo es exponencial en el caso peor, pero eficiente en la práctica.

Una defensa es que normalmente los algoritmos polinomiales tienen constantes pequeãs, mientras que los exponenciales son casi siempre realmente ineficientes.

Última observación: el *caso peor*. Usar el caso peor para hablar de la bondad de un algoritmo permite una matemática más fácil y facilita ciertas demostraciones. Sin embargo, el *caso medio* es en realidad una medida mucho mejor de la calidad de un algoritmo. Pero calcular el caso medio requiere estadística y muchas aproximaciones.

Capítulo 37

Jueves 12/05/11 (2 horas)

37.1 Complejidad abstracta

Vamos a ver (sin demostrarlos) hechos como los siguientes:

- aumentando los recursos de cierta forma se ensancha la clase de problemas resolubles (parece trivial, pero hemos visto que no siempre aumentar los recursos añade nuevos problemas);
- existen problemas arbitrariamente complejos;
- hay ciertas relaciones entre las clases de complejidad.

37.1.1 Funciones de medida apropiadas

Necesitamos una buenas definición de *función de medida*, es decir, las funciones que se usan para medir los recursos.

La idea es que si usamos una función de medida, ésta no puede ser ella misma demasiado compleja (es decir, se tiene que poder calcularla con recursos razonables). Digamos que $f(n)$ se tiene que calcular en $\mathcal{O}(f(n))$.

Definición 37.1 *$f : \mathbb{N} \mapsto \mathbb{N}$ calculable total es* apropiada *en inglés* proper*) si*

- *es monotona cresciente: $n \geq m \Rightarrow f(n) \geq f(m)$; y*
- *existe una MdT con k cintas (por algún k) tal que, para todo input x con $|x| = n$ (el tamaño del input), para con output "$*^{f(n)}$" (con $*$ símbolo especial) en tiempo $\mathcal{O}(n + f(n))$ y espacio $\mathcal{O}(f(n))$.*

Ejemplo 37.1 *Son apropiadas las funciones k, n, n^k, todos los polinomios, k^n, $n!$, (partes enteras de) $\log(n)$, $\sqrt{n}$. Si f y g son apropiadas, también lo son $f + g$, fg y $f \circ g$.*

37.1.2 Jerarquías entre clases

Bajo ciertas hipótesis, aumentar los recursos aumenta la clase de problemas resolubles. La inclusión (no estricta) es trivial; aquí hablamos de condiciones bajos las que la inclusión sí es estricta.

Teorema 37.1 *Si f es apropiada*

- $TIME(f(n)) \subset TIME(f(2n+1)^3)$
- $SPACE(f(n)) \subset SPACE(f(n)\log(f(n)))$

Un ejemplo de lo primero es con $f = n$ y los algoritmos de sorting. En este caso, f es apropiada y los algoritmos de sorting están en $TIME((2n+1)^3)$ pero no en $TIME(n)$.

En realidad la jerarquía es más densa. Lo que nos interesa aquí es que si $f(n)$ es un polinomio, también lo es $f(2n+1)^3$. Se demuestra lo siguiente:

Teorema 37.2

$$\mathcal{P} \subset \bigcup_{k \geq 1} TIME(2^{n^k}) = EXP$$

Demostración. Sigue porque

$$\mathcal{P} \subseteq TIME(2^n) \subset TIME((2^{2n+1})^3) \subseteq TIME(2^{n^2})$$

La primera inclusión sigue porque un exponencial crece más rápidamente que cualquier polinomio; la segunda viene del teorema anterior; y la tercera sigue porque $(2^{2n+1})^3 = 2^{6n+3}$ que es más pequeño que 2^{n^2}. El teorema vale porque EXP incluye $TIME(2^n)$ pero también $TIME(2^{n^2})$, $TIME(2^{n^3})$,.., $TIME(2^{n^{10}})$ etc. □

37.1.3 Complejidad según Blum

Es la base de la complejidad abstracta. Aquí se demuestran resultados que son independientes del modelo de cálculo. La teoría sólo se basa en dos axiomas.

Definición 37.2 *Una función ϕ es una* medida de complejidad *si devuelve un número natural para un input (ψ, x) (una función y su propio input), y*

- *$\phi(\psi, x)$ es definida si y sólo si lo es $\psi(x)$; y*
- *dado k, se puede decidir si $\phi(\psi, x) = k$.*

El siguiente teorema dice que existen problemas arbitrariamente difíciles, independientemente de los algoritmos usados.

Teorema 37.3 *Para toda función calculable total g existe un problema $L \in \Theta(f)$ (donde f es una medida de complejidad según la definición) con $f(n) \geq g(n)$* casi siempre *(es decir, excepto en un número finito de puntos).*

37.2 $\mathcal{P}$ y $\mathcal{NP}$

Vamos a caracterizar estas clases de complejidad a través de unos problemas *completos*, es decir, que están en ellas y son al menos igual de complejos que los demás.

Más precisamente, queremos encontrar, por ejemplo, un problema H que sea $\mathcal{NP}$-completo, es decir, tal que $H \in \mathcal{NP}$ y, para todo problema $L \in \mathcal{NP}$, tenemos que L se *reduce* a H.

Los problemas completas caracterizan una clase de complejidad porque expresan su estructura más profunda y la "esencia" de su dificultad.

Por ejemplo, los problemas H y H_1 que vimos en la parte de computabilidad son RE-completos para reducciones calculables totales (como las que usamos). Es decir, cualquier problema (lenguaje) RE se reduce a uno de ellos.

Normalmente, una clase de problemas es interesante si tiene problemas completos que sean interesantes desde el punto de vista compuacional, y que, por así decir, no se hayan creado artificialmente a partir de la definición de la clase. Por ejemplo, es el caso de RE, que tiene un problema completo H interesante *de por sí*: el problema de la parada.

Tenemos la siguiente jerarquía:

$$LOGSPACE \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq PSPACE = NPSPACE$$

También sabemos que $LOGSPACE \subset PSPACE$ (estricto). Lo que no sabemos es cuál de estas inclusiones es estricta:

$$LOGSPACE \subseteq \mathcal{P} \qquad \mathcal{P} \subseteq \mathcal{NP} \qquad \mathcal{NP} \subseteq PSPACE$$

La relación entre $\mathcal{P}$ y $\mathcal{NP}$ es especialmente interesante. Una reducción que nos sirva de algo no tiene que ser demasiado "difícil"; por ejemplo, no nos valdría (como en computabilidad) decir que π tiene que ser simplemente recursiva, porque si no cualquier algoritmo exponencial o incluso más complejo se podría reducir a uno linear o constante. Hace falta una reducción que sea *polinomial en tiempo* o bien *logarítmica en espacio*.

Definición 37.3 *L se reduce* eficientemente *a L' (escrito $L \leq_{LOGSPACE} L'$ si existe una π que opera en espacio logarítmico tal que, para todo x, $x \in L$ si y sólo si $\pi(x) \in L'$.*

Están funciones π son fáciles porque vale lo siguiente:

Teorema 37.4

$$f \in LOGSPACE \Rightarrow f \in \mathcal{P}$$

Demostración. Existe una MdT que calcula f usando $\log(|x|)$ casillas para todo input x. El número de configuraciones posible para dicha máquina es

$$\#Q \times \#\Sigma^{\log(|x|)} \times \log(|x|)$$

donde Q es el conjunto de estados y Σ es el conjunto de símbolos; el segundo factor dice cuantas cosas distintas pueden estar escritas en la cinta; el tercer factor tiene en cuenta la posición de la cabeza. Una compuatación que termina no puede pasar dos veces por la misma configuración, por lo que el número arriba limita el numero de pasos de la computación. El teorema sigue porque el número es polinomial. □

Capítulo 38

Viernes 13/05/11 (2 horas)

38.1 Algunos problemas

38.1.1 SAT

Introducción: elementos de lógica proposicional.

38.1.2 HAM

- reducibilidad de HAM a SAT (demostración informal)

38.1.3 CLIQUE

- reducibilidad de SAT a CLIQUE (sin demostración)

Bibliografia

[1] P. Blázquez. Estudio de los sistemas de descubrimiento científico automatizado. Master's thesis, Fi, UPM.

[2] G. S. Bollos and R. C. Jeffrey. *Computability and Logic, 3rd Edition*. Cambridge University Press, 1994.

[3] A. Church. A note on the entscheidungsproblem. 1936.

[4] S. B. Cooper. *Computability Theory*. Chapman & Hall/CRC, 2004.

[5] N.J. Cutland. *Computabilility*. Cambridge University Press, 1980.

[6] M. Davis, editor. *The Undecidable*. Raven Press, 1965.

[7] K. Gödel. Sobre proposiciones formalmente indecidibles de los principia mathematica y sistemas afines. 1931.

[8] D. Hofstadter. *Gödel, Escher, Bach*.

[9] J. Hopcroft, R. Motwani, and J. Ullman. *Introducción a la Teoría de Autómatas, Lenguajes y Computación*. Addison-Wesley, 2002.

[10] H. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1997.

[11] Nagel, Newman, Gödel, and Girard. *Le théorème de Gödel*. Eds. de Seuil, 1958.

[12] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[13] P. Odifreddi. *Classical Recursion Theory*. North Holland, 1989.

[14] R. Penrose. *The Emperor's New Mind*. Oxford University Press, 1990.

[15] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[16] Simon. Explaining the ineffable. In *Proceedings of 14th IJCAI*, 1995.

[17] A. Turing. On computable numbers, with an application to the entscheidungsproblem. 1936 (1937).

[18] A. Turing. Computing machinery and intelligence. *MIND*, 1950.