1

Cálculo simbólico

Índice General

1.1 La información y el lenguaje. Introducción

1. *Informática*: conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores. *Computadora*: máquina capaz de aceptar unos datos de entrada, efectuar con ellos operaciones lógicas y aritméticas, y proporcionar la información resultante a través de un medio de salida; todo ello sin intervención de un operador humano y bajo el control de un programa de instrucciones previamente almacenado en el propio computador.

Informática

Computadora

2. En lingüística, se caracteriza la *información* por sus propiedades: (1) aporta un conocimiento desconocido al receptor; (2) es identificable; (3) es útil. Cuando un

información

individuo adquiere información, lo hace en virtud de una petición de conocimiento cuyo cometido es obtener un conocimiento desconocido e inteligible que satisfaga convenientemente la demanda.

3. La información y la representación de la información son dos cosas diferentes. La computadora es la entidad agente que realiza un procesamiento de la representación de cierta información que se le proporciona como entrada produciendo una nueva representación que interpretada como información da respuesta a cierto problema para el que se diseño dicho procesamiento

La representación de esta información se lleva a cabo mediante un simbolismo dentro de un marco abstracto. Para ello hay que disponer de un método sistemático que nos permita representar el universo de objetos combinando un conjunto finito y no muy grande de símbolos (alfabeto).

En ciertos lenguajes la información se representa de forma forma altamente estructurada, con reglas de codificación: léxica, sintáctica y semántica.

4. Algunos lenguajes son útiles para expresar conceptos de la forma más exacta y concisa posible, y también como herramientas para procesar información mediante la transformación (*reescritura*) de la representación de esta información (cadenas de símbolos) según ciertas reglas de reescritura.

reescritura

Los lenguajes formales se diseñan de modo que mediante la manipulación sintáctica de una frase (*fórmula*) que expresa una proposición verdadera, podamos obtener nuevas proposiciones verdaderas desconocidas. Sistemas muy conocidos son la lógica proposicional, la lógica de predicados y la aritmética.

Al sistema formado por un lenguaje formal, y un conjunto de reglas que nos permiten escribir otras nuevas fórmulas, o averiguar si al menos son equivalentes, plasmando algún tipo de razonamiento, lo denominamos *cálculo*.

cálculo

proceso informático

5. Un *proceso informático* es una máquina dedicada a recibir por la entrada secuencias de símbolos y obtener a la salida una nueva secuencia de símbolos resultado de una transformación, o reescritura de la cadena original, siguiendo un conjunto de reglas preestablecidas, en un orden determinado. La tarea del informático es diseñar y comprender tales sistemas.

1.2 Cadenas, alfabetos y lenguajes

símbolo

6. Un *símbolo* es una entidad abstracta sin definición formal. Una cadena (o palabra) es una secuencia finita de símbolos yuxtapuestos. La longitud de una cadena w ($|w|$), es el número de símbolos de que consta la cadena. La cadena vacía (ϵ), es la cadena que consta de cero símbolos ($|\epsilon| = 0$).

7. La concatenación de dos cadenas es la cadena formada escribiendo la primera seguida de la segunda, sin espacios entre medias. La cadena vacía es el elemento neutro para el operador de concatenación. Es decir, $\epsilon w = w\epsilon = w$ para cualquier cadena w.

alfabeto
lenguaje

8. Un *alfabeto* es un conjunto finito de símbolos. Un *lenguaje* es un conjunto de cadenas de símbolos tomados de algún alfabeto. El conjunto vacío $\emptyset$, y el conjunto

que consiste en la cadena vacía $\{\epsilon\}$ son lenguajes, siendo ambos diferentes.

1.3 Lenguajes formales para el cálculo

1.3.1 Expresiones

9. Las expresiones son cadenas de símbolos sobre un alfabeto como constantes, variables, y operadores. La notación matemática habitual sigue el siguiente esquema recursivo, "son expresiones...

- constantes y variables,
- cosas como tipo (E), donde E es una expresión,
- cosas como tipo $\circ E$, donde $\circ$ es un operador unario prefijo y E es una expresión,
- cosas como $D \star E$, donde $\star$ es un operador binario infijo y D y E son expresiones,

En las expresiones, los paréntesis indican agregación. De forma que, $2 \cdot (3 + 5)$ denota el producto de 2 y $3 + 5$. Para reducir el uso de paréntesis se suele asignar precedencia a los operadores.

10. Cada variable que aparece en una expresión puede instanciarse en uno de los valores de un conjunto. Con las variables generalizamos las expresiones sobre conjuntos de estados. Un *estado* o *asignación* es una relación de variables y valores. Después de remplazar todas las variables de una expresión por sus valores correspondientes en ese estado, obtenemos una expresión donde solo aparecen constantes. En este punto, se puede evaluar la expresión, si es el caso, para obtener el valor resultante. valor de la expresión resultante.

estado
asignación

1.3.2 Substitución textual

11. Sea E y R expresiones y sea x una variable. Empleamos la notación

$$E[x := R] \quad \text{o} \quad E^x_R$$

para denotar una expresión que es lo mismo que E pero con todas las apariciones de x remplazadas por "(R)". El acto de remplazar todas las apariciones de x por "(R)" en E se denomina *substitución textual*.

substitución textual

12. Regla de inferencia de la substitución — la substitución textual puede expresarse en forma de *regla de inferencia*, que provee un mecanismo sintáctico para derivar "verdades". Una *regla de inferencia* consiste en una lista de expresiones, denominadas *premisas* o *hipótesis*, sobre una línea y una expresión denominada *conclusión* debajo.

regla de inferencia

(1.1) **Substitución:** $\dfrac{E}{E[v := F]}$

1.3.3 Substitución textual e igualdad

cierto
falso

13. Al evaluar $X = Y$ en un estado, resulta *cierto* si X e Y tienen el mismo valor y resulta *falso* si tienen valores diferentes. De esta forma introducimos en el lenguaje un nuevo símbolo "=". La igualdad se caracteriza por las siguientes leyes:

(1.2) **Reflexividad:** $X = X$

(1.3) **Simetría:** $(X = Y) = (Y = X)$

(1.4) **Transitividad:** $\dfrac{X=Y, Y=Z}{X=Z}$

(1.5) **Leibnitz:** $\dfrac{X=Y}{E[z:=X]=E[z:=Y]}$

En nuestra formulación, las expresiones (1.2) y (1.3) adoptan la forma de axiomas, mientras que (1.4) y (**??**) toman la forma de reglas de inferencia.

La regla de Leibnitz (1.5) viene a decir que dos expresiones son iguales en todos los estados si y solo si al reemplazar una por la otra en cualquier expresión E no cambia el valor de E (en cualquier estado).

1.3.4 Regla de Leibnitz y evaluación de funciones

14. Una función es una regla para calcular un valor resultante a partir de otros, que se dan como dato, y se denominan argumentos. A la función se le atribuye un nombre y va seguida de su argumento, separada por ".".

Si se ha definido la función g como:

(1.6) $g.z : E$

se define la aplicación de función $g.X$ para cualquier argumento X como $g.X = E[z := X]$.

15. La regla de Leibnitz (1.5) relaciona la igualdad y la aplicación de función:

(1.7) **Leibnitz:** $\dfrac{X = Y}{g.X = g.Y}$.

1.3.5 Razonando utilizando la regla de Leibnitz

16. La regla de Leibnitz (1.5) nos da da un método para demostrar que dos expresiones son iguales. En este método, el formato que empleamos para mostrar una aplicación de Leibnitz es

$$\begin{array}{ll} & E[z := X] \\ = & \quad \langle X = Y \rangle \\ & E[z := Y] \end{array}$$

La primera y terceras líneas son las expresiones de igualdad de la conclusión en Leibnitz; la sugerencia del medio es la premisa $X = Y$. La sugerencia está indentada y delimitada por $\langle$ y $\rangle$. La variable z de Leibnitz no se menciona.

1.4 Lógica

17. Una lógica es un lenguaje formal dotado de un mecanismo deductivo. El lenguaje está formado por un conjunto de expresiones denominadas fórmulas bien formadas constituidas por elementos de un conjunto de símbolos que representan variables y un conjunto de símbolos denominados constantes lógicas que permiten formar expresiones compuestas a partir de fórmulas elementales.

A los conjuntos anteriores, que forman el alfabeto del lenguaje se unen las reglas de formación que permiten obtener nuevas fórmulas y deducir la formación de fórmulas compuestas.

18. El mecanismo deductivo de una lógica se compone de axiomas y de reglas de inferencia. Los axiomas son fórmulas particulares que representan los principios lógicos fundamentales, o así se les considera (principio de no contradicción, de exclusión de medios). Los axiomas se consideran fórmulas verdaderas.

Las reglas de inferencia, preservando la caracterización expresada por los axiomas, aseguran la parte creativa del mecanismo deductivo, permitiendo obtener nuevas fórmulas verdaderas a partir de otras fórmulas verdaderas o de los axiomas (Leibnitz, *modus ponens*,...

1.4.1 Teorema, y demostración formal

19. Un *teorema* es o un axioma o una fórmula deducida por la aplicación de las reglas de inferencia a partir de los axiomas. Se expresa: "$\vdash A$". teorema

Una *demostración* de un teorema A es una serie de fórmulas $A_1, \ldots, A_i, \ldots, A$ donde A_i es un axioma, o bien se obtiene por la aplicación de una regla de inferencia con los teoremas A_j y A_k producidos anteriormente, donde $k < i$ y $j < i$. demostración

20. Una lógica es *consistente* si al menos una de sus fórmulas es un teorema y al menos una no lo es; en caso contrario la lógica es inconsistente. consistente

1.4.2 Semántica (Modelos)

21. El dominio de discurso es el área de aplicación de algún sistema axiomático y sobre el que se interpretan las fórmulas. La *interpretación* es una asignación de significado a los operadores, constantes por una parte, y variables de una lógica por otra, que de otro modo carecerían de aplicación práctica. Al interpretar las variables se atribuyen valores a éstas, lo que describe un estado concreto. interpretación

22. Sea S un conjunto de interpretaciones de una lógica y F una fórmula de la lógica. F es *satisfacible* bajo S si y solo si al menos una interpretación de S evalúa F a *cierto*. F es *válido* bajo S si y solo si cada interpretación de S evalúa F a *cierto*. En otras palabras: F es satisfacible si y solo si F se evalúa a *cierto* en al menos un estado. F es válido si y solo si F se evalúa a *cierto* en todos los estados. satisfacible válido

23. Una interpretación M es un *modelo* de una lógica si y solo si cada teorema A se evalúa a *cierto* en esa interpretación, $M \models A$. modelo

24. Una lógica es sólida si y solo si cada teorema es válido. Una lógica es completa si y solo si cada fórmula válida es un teorema.

La solidez quiere decir que todos los teoremas de la lógica son sentencias verdaderas sobre el dominio del discurso. La completitud quiere decir que toda fórmula válida puede demostrarse. La carencia de completitud en una lógica no es un síntoma de debilidad, de hecho Gödel demuestra que cualquier sistema lógico formal más complicado que el cálculo de proposiciones es necesariamente incompleto si es sólido.

decidible

25. Una lógica es *decidible* si existe un procedimiento que permite computar si una fórmula F es cierta en cualquier estado del espacio de estados definido por los posibles valores de las variables de F; tal procedimiento se denomina *procedimiento de decisión*.

procedimiento de decisión

Dicho de otro modo, una lógica es decidible si podemos calcular si una fórmula es un teorema, y en el caso de que la lógica sea sólida será válida. La lógica de proposiciones es decidible. La aritmética de enteros es un caso de sistema sólido pero no decidible.

2

Lógicas

Índice General

2.1 Lógica de proposiciones

2.1.1 Sintaxis y evaluación de expresiones booleanas

26. Las expresiones booleanas se construyen con las constantes *cierto* y *falso* (valores booleanos), asociándolas por medio de funciones booleanas $\equiv$, $\not\equiv$, $\neg$, $\vee$, $\wedge$, $\Rightarrow$ y $\Leftarrow$. Se dice que las expresiones booleanas son de *tipo booleano*.

27. Estas funciones booleanas se suelen escribir en forma infija, por eso se denominan operadores booleanos o conectivas booleanas.

$\neg a$ — $\neg$ es la negación o función *not*, que invierte el valor de verdad de a.

$a = b$ — $=$ es la igualdad convencional. Es formalmente equivalente a la equivalencia $\equiv$, cuyo uso se reserva para el caso de operandos booleanos.

$a \neq b$ — $\neq$ es la desigualdad convencional. El mismo comentario anterior se aplica a $\not\equiv$. A menudo se denomina también *xor* que representa la función *o exclusivo*.

$a \vee b$ — $\vee$ se denomina *disyunción* o función *or*, que es cierto si y solo si alguno de sus operandos son ciertos.

$a \wedge b$ — $\wedge$ se denomina *conjunción* o función *and*, que es cierto si y solo si ambos operandos son ciertos.

$a \Rightarrow b$ — $\Rightarrow$ se denomina implicación, a antecedente y b consecuente, y es falso si y solo si b resulta ser falso cuando a es cierto.

$a \Leftarrow b$ — $\Leftarrow$ se denomina consecuencia, a consecuente y b antecedente. Se usa con el fin de simplificar ciertas demostraciones dado que $a \Leftarrow b$ es igual a $b \Rightarrow a$.

28. La forma más directa de evaluar expresiones booleanas es mediante la tabla de valores de verdad, o tablas veritativas, donde se evalúa la expresión para cada uno de los posibles valores que pueden adoptar los operandos de la expresión.

2.1.2 Satisfacibilidad, validez y dualidad

29. Una expresión booleana P se *satisface* en un estado si su valor es cierto en ese estado; P es *satisfacible* si existe un estado en el que se satisface; P es *válida* si se satisface en cualquier estado, también se dice que es una *tautología*.

satisfacible

dual

30. El *dual* P_D de una expresión booleana P se construye a partir de P intercambiando las apariciones de *cierto* y *falso*; $\wedge$ y $\vee$; $\equiv$ y $\not\equiv$; $\Rightarrow$ y $\not\Leftarrow$; $\Leftarrow$ y $\not\Rightarrow$.

Principio de Dualidad.

- Una expresión P es válida si y solo si $\neg P_D$ es válida.
- además, $P \equiv Q$ es válida si y solo si $\P_D \equiv Q_D$ es válida.

La última parte de este principio se conoce también como “ley de De Morgan”.

2.1.3 Proposiciones y su interpretación en castellano

31. La traducción de proposiciones del lenguaje natural en expresiones booleanas nos ayuda a identificar y resolver ambigüedades, y se aplica en el análisis, razonamiento, manipulación y simplificación de expresiones.

El mecanismo de traducción de una proposición p (variable booleana) en una expresión booleana sigue los siguientes pasos:

- Introducir variables booleanas para denotar subproposiciones.
- Remplazar estas subproposiciones por sus correspondientes variables booleanas.
- Traducir el resultado del paso anterior en expresiones booleanas, utilizando traducciones de palabras en castellano en sus "homólogos" operadores booleanos: "y" $\rightarrow \wedge$, "o" $\rightarrow \vee$, "no" $\rightarrow \neg$, "no es el caso" $\rightarrow \neg$, "si p entonces q" $\rightarrow p \Rightarrow q$.

32. Hay que averiguar si la conectiva "o" se utiliza de modo exclusivo o inclusivo. En el caso inclusivo "a o b" se representa por $a \vee b$. En el caso exclusivo por $a \not\equiv b$ o también $a \equiv \neg b$.

33. La implicación surge tanto de la aparición directa de la forma "si a entonces b", como de otras fórmulas, por ejemplo: "$a \not> 3 \;\Rightarrow\;$ *acabar programa*" se puede escribir como "$a > 3 \;\vee\;$ *acabar programa*".

También aparecen implicaciones de forma más retorcida como "Todo teléfono de la guía de Serrada está en la guía de Valladolid", se puede escribir como "Si un teléfono está en la guía de Serrada, está en la guía de Valladolid" ($gS \Rightarrow gV$).

34. Algunas frases con la construcción "si" no son propiamente implicaciones, sino equivalencias. Por ejemplo: "Si dos lados de un triángulo son iguales (li), el triángulo es isósceles(ti)" se traduce exactamente como $li \equiv ti$, no como $li \Rightarrow ti$, que es cierta si ocurre (al menos formalmente) que el triángulo sea isósceles sin que dos lados sean iguales.

35. Se puede codificar frases que contengan construcciones de "necesidad" y "suficiencia". En resumen "x es suficiente para y" se escribe $x \Rightarrow y$, "x es necesario para y" se escribe $y \Rightarrow x$, y "x es necesario y suficiente para y" se escribe $(x \Rightarrow y) \wedge (x \Leftarrow y)$. Esta última se puede reescribir como "x si y solo si y" o "x sii y", que se traduce como $x \equiv y$.

2.1.4 Axiomas del cálculo de proposiciones ecuacional

36. Un *cálculo* es un método o proceso de razonamiento mediante el cómputo con símbolos. En el cálculo de proposiciones se emplean variables proposicionales. Aquí se describe una formulación denominada lógica ecuacional **E**, y se orienta a la manipulación de expresiones booleanas. cálculo

37. Una parte importante, junto al cálculo, es el conjunto de *axiomas*, que son ciertas expresiones booleanas verdaderas que definen ciertas propiedades manipulativas. Otra parte importante son tres reglas de inferencia: Substitución (1.1), Transitividad (1.4) y Leibnitz (1.5). axiomas

38. Un *teorema* de este cálculo es o bien (i) un axioma o (ii) una expresión booleana que, empleando las reglas de inferencia, se demuestra que es igual a un axioma o teorema

a otro teorema demostrado anteriormente. La elección de un cierto conjunto de axiomas para nuestro cálculo puede conducir al mismo conjunto de teoremas que aparece en otros textos, que parten de otros axiomas. Incluso la forma de presentar los operadores booleanos puede ser diferente.

Todos los teoremas del cálculo proposicional **E** son válidos, y hay más, todas las expresiones válidas son teoremas del cálculo.

39. La equivalencia es asociativa y simétrica. Podemos presentar la constante *cierto* como equivalente a la equivalencia una proposición con sigo misma.

(2.1) **Asociatividad de $\equiv$:** $((p \equiv q) \equiv r) \equiv (p \equiv (q \equiv r))$

(2.2) **Simetría de $\equiv$:** $p \equiv q \equiv q \equiv p$

(2.3) **Identidad de $\equiv$:** $cierto \equiv q \equiv q$

40. La constante *falso* se define axiomáticamente, al igual que el comportamiento de la negación. La inequivalencia también se define por axioma sobre la negación y la equivalencia.

(2.4) **Definición de *falso* :** $falso \equiv \neg cierto$

(2.5) **Distributividad de $\neg$ sobre $\equiv$:** $\neg(p \equiv q) \equiv \neg p \equiv q$

(2.6) **Definición de $\not\equiv$:** $(p \not\equiv q) \equiv \neg(p \equiv q)$

41. La disyunción $\vee$ se define por los siguientes cinco axiomas.

(2.7) **Simetría de $\vee$:** $p \vee q \equiv q \vee p$

(2.8) **Asociatividad de $\vee$:** $(p \vee q) \vee r \equiv p \vee (q \vee r)$

(2.9) **Idempotencia de $\vee$:** $p \vee p \equiv p$

(2.10) **Distributividad de $\vee$ sobre $\equiv$:** $p \vee (q \equiv r) \equiv p \vee q \equiv p \vee r$

(2.11) **Medio excluido :** $p \vee \neg p$

Regla de oro

42. Para definir la conjunción $\wedge$ basta un axioma.

(2.12) **Regla de oro :** $p \wedge q \equiv p \equiv q \equiv p \vee q$

43. La implicación $\Rightarrow$se define también axiomáticamente, y la consecuencia $\Leftarrow$ a partir de la anterior.

(2.13) **Definición de Implicación $\Rightarrow$:** $p \Rightarrow q \equiv p \vee q \equiv q$

(2.14) **Consecuencia $\Leftarrow$:** $p \Leftarrow q \equiv q \Rightarrow p$

44. Se puede presentar una versión de la regla de equivalencia de Leibnitz mediante el operador $\Rightarrow$.

(2.15) **Leibnitz :** $(e = f) \Rightarrow (E^z_e = E^z_f)$ (E cualquier expresión)

2.1.5 Consistencia, completitud y solidez de E

45. Un sistema lógico formal (o lógica) es un conjunto de reglas definidas en términos de sendos conjuntos de símbolos, fórmulas, axiomas y reglas de inferencia. Para la lógica ecuacional **E** los símbolos son $(,)$, $=$, $\neq$, $\equiv$, $\not\equiv$, $\neg$, $\vee$, $\wedge$, $\Rightarrow$ y $\Leftarrow$, las constantes *cierto* y *falso* y las variables booleanas p, q, etc. Las fórmulas se construyen empleando esos símbolos. **E** tiene 15 axiomas y tres reglas de inferencia, y los teoremas son las fórmulas que puede demostrarse que son iguales a los axiomas mediante las reglas de inferencia.
46. La lógica **E** es consistente, porque **cierto** es un teorema y **falso** no lo es. Si se hubiera añadido el axioma "*falso* $\equiv$ *cierto*" **E** no sería consistente.
47. **E** es sólida y completa con respecto al sentido estándar de interpretación. Es muy fácil construir una lógica no sólida, basta con añadir un axioma insatisfacible, como $p \wedge \neg p$ en el caso de **E**.
48. Se define el procedimiento de decisión para la lógica **E** como: Computar la interpretación de una fórmula F en cada estado posible del espacio de estados definido por las variables booleanas en F. F es un teorema de **E** sii se evalúa a *cierto* en cada estado. Dado que existe un procedimiento de decisión para **E**, se dice que es decidible.

2.2 Lógica de predicados de primer orden

2.2.1 Predicados y cálculo de predicados. Definición de Teoría

49. El cálculo de predicados define una clase más general de razonamientos que el cálculo de proposiciones. Una fórmula de cálculo de predicados es una expresión booleana en la cual algunas variables booleanas pueden haber sido remplazadas por Predicados o cuantificadas universal y existencialmente. El cálculo de predicados puro, incluye los axiomas del calculo proposicional, junto con axiomas para las cuantificaciones $(\wedge x|\ R : P)$ y $(\vee x|\ R : P)$.
50. Un *predicado* es una aplicación de una función booleana cuyos argumentos pueden ser de otro tipo diferente a $\mathbb{B}$ (p.ej.:*cierto*, *falso*). Los nombres de estas funciones se denominan *símbolos de predicado*. Formalmente, estos símbolos no reciben ninguna interpretación (excepto la igualdad '='), solo reglas para manipular expresiones formadas con ellos: el cálculo de predicados es sólido. predicado
51. Se puede construir una *teoría* apartir del cálculo de predicados, dando significado a algunos de los símbolos de función no interpretados ya. Un ejemplo es la *teoría de números enteros* que incluye funciones para manipular números $+$, $-$, $\cdot$, $<$, $>$, etc. otro la *teoría de conjuntos* que añade axiomas para manipular $\in$, $\cup$ y $\cap$. teoría

2.2.2 Cuantificación: Universal y Existencial

52. La cuantificación universal se escribe convencionalmente $(\forall x|R : P)$ y se lee "para todo x que cumple R (Rango), P (Predicado) es cierto". El símbolo $\forall$ se deno-

mina *cuantificador universal*. Para éste, se definen los siguientes axiomas importantes:

(2.16) **Intercambio :** $(\forall x|R : P) \equiv (\forall x| : R \Rightarrow P)$

(2.17) **Distribución de $\vee$ en $\forall$:** $P \vee (\forall x|R : Q) \equiv (\forall x|R : P \vee Q)$

Si en la última expresión, x no aparece en P.

53. En una expresión booleana con variables, se dice que éstas variables están libres, y la expresión está *abierta*; su valor depende del estado. Cuando cuantificamos universalmente sobre una expresión booleana, con respecto a una variable concreta, se dice que está ligada. Si todas las variables de una expresión están cuantificadas universalmente, la expresión está *cerrada*, y la cuantificación se llama *cierre* o *cerradura*.

variable libre

variable ligada

cierre, cerradura

El siguiente metateorema caracteriza el cierre de un teorema.

(2.18) **Metateorema :** P es un teorema sii $(\forall x| : P)$ es un teorema.

54. Hay teoremas muy interesantes sobre la cuantificación universal, entre ellos están:

(2.19) **Fortalecimiento/debilitamiento de rango :**

$$(\forall x|Q \vee R : P) \Rightarrow (\forall x|Q : P)$$

(2.20) **Fortalecimiento/debilitamiento de cuerpo :**

$$(\forall x|R : P \wedge Q) \Rightarrow (\forall x|R : P)$$

(2.21) **Monotonicidad de $\forall$:**

$$(\forall x|R : Q \Rightarrow P) \Rightarrow ((\forall x|R : Q) \Rightarrow (\forall x|R : P))$$

(2.22) **Ejemplarización de $\forall$:**

$$(\forall x| : P) \Rightarrow P[x := E]$$

55. La cuantificación existencial se escribe convencionalmente $(\exists x|R : P)$ y se lee "existe un x que cumple R, para el que P es cierto". El símbolo $\exists$ se denomina *cuantificador existencial*. El valor de x para el que se el predicado cuantificado es cierto $(R \wedge P)[x := \hat{x}]$se denomina *testigo*.

56. El siguiente axioma relaciona la cuantificación existencial con la universal, y se denomina axioma Generalizado de De Morgan.

(2.23) **Generalizado de De Morgan :** $(\exists x|R : P) \equiv \neg(\forall x|R : \neg P)$

57. Para el cuantificador existencial también tenemos teoremas muy interesantes:

(2.24) **Fortalecimiento/debilitamiento de rango :**

$$(\exists x|R : P) \Rightarrow (\exists x|Q \vee R : P)$$

(2.25) **Fortalecimiento/debilitamiento de cuerpo :**

$$(\exists x|R : P) \Rightarrow (\exists x|R : P \vee Q)$$

(2.26) **Monotonicidad de $\exists$:**

$$(\forall x|R : Q \Rightarrow P) \Rightarrow ((\exists x|R : Q) \Rightarrow (\exists x|R : P))$$

(2.27) **$\exists$-Introducción :**

$$P[x := E] \Rightarrow (\exists x| : P)$$

(2.28) **Intercambio de cuantificadores :**

$$(\exists x|R : (\forall y|Q : P)) \Rightarrow (\forall y|Q : (\exists x|R : P))$$

En este último caso, si las variables y y x no aparecen en R y Q respectivamente.

2.2.3 Predicados y su interpretación en castellano

58. El cálculo de proposiciones nos permite razonar empleando proposiciones lógicas donde se detallan propiedades y relaciones entre individuos. Este cálculo permite formalizar aspectos que escapan a la lógica de proposiciones.
59. El cuantificador universal $\forall$ aparece en las frases bajo las formas: *todo, cada uno, cualquier, para todo*. El cuantificador existencial aparece bajo las formas: *existe, algún, hay un, al menos un, para algún*.
60. Los individuos, no aparecen directamente en la expresión, sino como argumento de cierta proposición que describe una propiedad del individuo, que puede ser cierta o falsa. Estas funciones proposicionales pueden tener 0, uno, o más argumentos.

2.2.4 Lógica de predicados y programación

61. La lógica de predicados se emplea con éxito en la especificación formal de programas imperativos. Una "tripleta de Hoare" es $\{Q\}\ S\ \{R\}$, donde S es un programa, Q la precondición y R la postcondición, y se interpreta: *Al ejecutarse S partiendo de un estado en que Q es cierto, termina y conduce a un estado en que R es cierto*.
62. El cálculo de predicados permite especificar conjuntos de estados de un cómputo. En particular, una especificación puede ser no determinista. En el repertorio del programador aparecen funciones como: *asignación*, *condicional* y *bucle*, con las que se describe el programa. Para verificar que un programa cumple sus especificaciones se realiza una demostración formal que partiendo de cada axioma de definición de la operación, y la post y pre condición, conduce a la especificación.
63. El axioma de la función de asignación $x := E$ se define (salvando los problemas de dominio de definición de las expresiones) como:

(2.29) **Asignación :** $\{R[x := E]\}\ x := E\ \{R\}$

64. La sentencia condicional, tiene la siguiente forma en muchos lenguajes imperativos:

(2.30) **Condicional :** **if** B **then** S_1 **else** S_2

donde si la expresión booleana B es *cierta*, se ejecuta la sentencia S_1, de otro modo se ejecuta la sentencia S_2. Este es el caso de una sentencia altenativa *determinista*, si la lógica es sólida (cuando B es falso $\neg B$ es cierto).

Para demostrar $\{Q\}\ IF\ \{R\}$, es suficiente probar $\{Q \wedge B\}\ S_1\ \{R\}$ y $\{Q \wedge \neg B\}\ S_2\ \{R\}$.

2.3 Ampliaciones de la lógica de predicados

65. La lógica permite modelar los procesos de razonamiento, y existen sistemas de razonamiento que no se pueden modelar con la lógica de proposiciones ni la logica de predicados. Las alternativas se pueden clasificar en dos grupos: (a) las lógicas en las que se amplía el cálculo de proposiciones y predicados, y (b) otras lógicas que contienen postulados incompatibles, como la lógica multivalorada o la lógica borrosa.

2.3.1 Lógica de predicados de orden superior

66. En la lógica de predicados de segundo orden, los predicados pueden cuantificarse y utilizarse como variables. Esto viene a decir que las funciones que se utilizan para definir propiedades de los elementos, y que permiten definir conjuntos de elementos que comparten las mismas propiedades, también están incluidos en conjuntos, cuyos elementos son propiedades. Podemos escribir, por ejemplo, $(\exists R| : (\forall x|E : R(x)))$. De similar modo se llega a la lógica de predicados de tercer orden, etc.

67. Aunque la sintáxis es sencilla de ampliar, no ocurre lo mismo con el sistema axiomático, que se presta a inconsistencias y paradojas. En programación, un ejemplo de lógica de orden superior, sería la *teoría de tipos*, donde éstos pueden ser operandos de una lógica de segundo orden.

2.3.2 Lógica de clases y lógica de relaciones

68. La lógica de clases y la lógica de relaciones son dos formulaciones alternativas a dos casos particulares del cálculo de predicados.

69. En cuanto a la lógica de clases, se parte de la noción de clase, como entidad abstracta que designa los individuos que comparten una propiedad. Si las operaciones entre clases coinciden con las operaciones del álgebra de conjuntos, y éste a su vez, es un álgebra de Boole, podríamos definir la lógica de clases, como una lógica de proposiciones de segundo orden. En esta, los operadores sobre clases (A, B, etc.) son ($\cup$, $\cap$, $\in$, $\subset$, etc), juntamente con los operadores booleanos. Puesto que a toda propiedad corresponde una clase, y los predicados monádicos representan propiedades de individuos, todo lo que pueda representarse en lógica de predicados monádicos puede también representarse en lógica de clases y estudiarse mediante el cálculo de proposiciones.

70. Del mismo modo que en el caso anterior, la lógica de relaciones es el equivalente a la lógica de predicados poliádicos (o de *aridad* n) En el álgebra de relaciones, los predicados se definen como pares ordenados de elementos.

71. En el álgebra de relaciones se define la relación universal y la relación vacía, junto con un conjunto de operaciones: complemento, unión, suma, diferencia, selección, proyección y productos de relaciones. Son la base para el diseño de un tipo de base de datos denominado *base de datos relacional*.

base de datos relacional

2.3.3 Lógica modal

72. La lógica modal se introduce para dar cuenta de las llamadas "expresiones modales" o "modalidades". El origen de estas se halla en la discursión sobre las nociones de implicación material e implicación estricta. Para C. Lewis la *implicación material* es la operación habitual denotada por el símbolo $\Rightarrow$ del cálculo de proposiciones (exceptuando la lógica intuicionista). De su utilización aparecen paradojas como $A \Rightarrow (B \Rightarrow A)$ y $\neg A \Rightarrow (A \Rightarrow B)$, ambas teoremas.

implicación material

La primera permite deducir que tomando como premisa A, la consecuencia es cualquier implicación cuya consecuencia sea A. La segunda, que si $\neg A$ es la premisa, la consecuencia es cualquier implicación cuya premisa sea A.

73. Para resolver el problema, se introduce la *implicación lógica* o estricta $\Longrightarrow$, que se define empleando una conectiva especial denominada *posibilidad* $\Diamond$:

implicación lógica

$$(A \Longrightarrow B) \triangleq \neg\Diamond(A \wedge \neg B)$$

que se lee:"Es imposible que A sea verdadera y B falsa, ya que A implica estrictamente B".

74. El desarrollo de estas ideas condujo a la formulación de la lógica modal. La más conocida es la formulación donde $\Diamond$ significa "posibilidad" y $\Box$ "necesidad", aunque se permiten otros modos, para expresar el conocimiento (lógica epistémica) la obligación (lógica deóntica) o el tiempo (lógica temporal). Además existen variedad de formulaciones en función de los axiomas, conocidas como S1, S2, ... Su ámbito de aplicación es variado, abarcando sistemas inferenciales no monotónicos, teoría de lenguajes de programación, concurrencia, etc...

75. Una noción útil en este ámbito, es la de "mundos posibles", que es el dominio del discurso de todos las posibles configuraciones de estados, donde se emplea un teorema. $\Diamond A$ significa que A puede ser válido en alguna configuración o "mundo", $\neg\Diamond A$ que A no puede ser válido en ningún mundo, $\Box A$ que es válido en cualquier mundo. Según esto, $\Diamond p \equiv \neg\Box\neg p$, $\Box p \equiv \neg\Diamond\neg p$ y $\neg\Diamond p \equiv \Box\neg p$.

2.3.4 Lógica temporal

76. La lógica temporal es un tipo de lógica modal, donde el modo es el tiempo. En ella, la función de interpretación se asocia a un instante para interpretar un predicado. Además se añaden dos predicados temporales $F(A)$ que se lee "A ocurrirá en

algún instante", y $P(A)$ que se lee "A ocurrió en algún instante" y un predicado de precedencia temporal.
77. Las operaciones de necesidad y posibilidad se aplican a predicados, construyendo sentencias donde se expresan situaciones como "posiblemente ocurrirá A", "ocurrirá A" ó "posiblemente nunca ocurrirá A", etc. La lógica temporal tiene interés en especificación y verificación de programas concurrentes, planificación, etc. En éstas áreas se emplean diversas perspectivas del tiempo: lineal vs. ramificada y continua, semicontinua y discreta.

2.4 Lógicas multivaloradas

78. Mediante las lógicas multivaloradas se pueden expresar razonamientos en los que hay cabida para proposiciones que toman valores distintos de la certeza y la falsedad. A nivel pragmático, se puede utilizar para expresar incertidumbre, indecisión, o posibilidad. La más habitual es la lógica trivalorada (o trivalente).
79. La lógica probabilitaria (o probabilística), emplea un dominio de discurso donde las expresiones booleanas pueden tomar un valor en un rango, y las conectivas plasman en la realidad operaciones de asociación de probabilidades a proposiciones, en el marco de la Teoría de Probabilidades. Este es un caso de lógica infinítamente valorada.

2.5 Lógica borrosa

80. En la lógica borrosa, se plantea la posibilidad, no solo de que el dominio de valores de las proposiciones es multivaluado, sino que la evaluación misma de las expresiones lógicas lleva asociado un valore de incertidumbre. Si podemos representar los diferentes valores que puede tomar una variable proposicional como la clase a la que se asocia esa variable al evaluarla, la lógica borrosa contempla que la relación de pertenencia a esa clase venga caracterizada por una magnitud que expresa la *similitud* a la clase.
81. Los conjuntos que emplea la lógica borrosa se pueden denominar *borrosos* o *difusos*, para indicar que no existe un criterio que determine exactamente un límite entre pertenencia y no pertenencia al conjunto. Dado un universo U, un subconjunto borroso $\underline{A}$, de U, es un conjunto de pares

$$\{(x|\mu_A(x))\}, \forall x \in U$$

donde $\mu_A(x)$ es una función que toma sus valores en un conjunto M llamado *conjunto de pertenencia*, en la teoría clásica de conjuntos $\{0, 1\}$
82. Mediante la lógica borrosa se puede establecer un álgebra de relaciones borrosas, tanto entre conjunto ordinarios, como borrosos. La aplicación de la lógica borrosa ("fuzzy logic") abarca el ámbito de los sistemas expertos donde se trabaja con incertidumbre, mecanismos de aprendizaje y sistemas de control robusto.

3

Funciones: Cálculo-λ

Índice General

3.1 Introducción

83. El cálculo-λ es una teoría de funciones, desarrollada por el lógico Alonzo Church, como un fundamento para las matemáticas. Mediante él se trata de formalizar la forma en que se pueden escribir funciones, y reescribirlas de otra forma,

de modo que sean equivalentes. A partir de un número muy primitivo de funciones se puede generar cualquier función computable.

84. Se puede expresar el significado de programas imperativos empleando cálculo-λ. Programar empleando lenguajes que implementen, en la práctica el cálculo-λ supone construir una función. Esta función tomará los datos de entrada de un problema, y proporcionará los datos de salida. La programación en estos lenguajes, conlleva ciertas ventajas y desventajas con respecto a la programación habitual de los lenguajes procedimentales.

85. Alonzo Church desarrolla el cálculo-λ en la década de 1930. En ésta década, Haskell Curry redescubre y extiende la teoría de Moses Schönfinkel sobre funciones llamadas "combinadores" y muestra que son equivalentes. Por aquella época Kleene muestra que el cálculo-λ era un sistema de cómputo universal. En la década de 1950, John McCarthy inventa el lenguaje LISP, bajo la influencia del cálculo-λ. Poco más tarde, Peter Landin muestra cómo se puede traducir el significado de los lenguajes imperativos en cálculo-λ e inventa ISWIM, un lenguaje de programación de prototipos que influye fuertemente en el diseño de lenguajes imperativos y funcionales.

86. El desarrollo del cálculo-λ ha traído consigo importantes desarrollos en la teoría de los lenguajes de programación, como: (i) el estudio de cuestiones fundamentales sobre computación; (ii) el diseño de lenguajes de programación; (iii) la semántica de los lenguajes de programación; (iv) la arquitectura de computadoras.

3.2 Sintaxis y semántica del cálculo-λ

expresiones-λ

87. El cálculo-λ es una notación para definir funciones. Las expresiones de la notación se denominan expresiones-λ y cada una de esas expresiones denota una función. Hay tres clases de expresiones-λ: variables, aplicaciones de funciones o combinaciones y abstracciones.

variables

88. Las *variables* se denotan por letras como x, y, z, etc. El valor de las funciones que representan las variables dependen de aquello a lo que estén ligadas en el *entorno* de la variable. La ligadura ("binding") se lleva a cabo por las abstracciones. Los valores concretos de las variables se representa por V_1, V_2, etc.

aplicación de función

89. Si E_1 y E_2 son expresiones-λ, también lo es $(E_1\ E_2)$. Esta expresión representa el resultado de la *aplicación de función* de la función denotada por E_1 sobre la función denotada por E_2.

90. Si V es una variable y E es una expresión-λ, $(\lambda V.E)$ es una abstracción con *variable ligada* V y *cuerpo* E. Tal abstracción denota la función que toma un argumento a y regresa el resultado de evaluar E en un entorno en el cual la variable ligada V representa a. De forma más específica, la abstracción $\lambda V.E$ representa una función que toma un argumento E' y lo transforma en el objeto representado por $E[V := E']$.

3.2.1 Simplificación de la sintaxis del cálculo-λ y otras convenciones

91. Empleando la notación BNF, la sintaxis de una expresión-λ es:

$$\begin{aligned} \langle \textit{expresión} - \lambda \rangle &::= \langle \textit{variable} \rangle \\ &::= (\langle \textit{expresión} - \lambda \rangle \langle \textit{expresión} - \lambda \rangle) \\ &::= (\lambda \langle \textit{variable} \rangle . \langle \textit{expresión} - \lambda \rangle) \end{aligned}$$

92. Para simplificar la notación se emplean las siguientes normas:

1. La aplicación de función se asocia desde la izquierda, de modo que $E_1\, E_2 \ldots E_n$ representa $((\ldots (E_1\, E_2) \ldots) E_n)$.
2. $\lambda V.E1\, E2 \ldots E_n$ representa $(\lambda V.(E_1\, E_2 \ldots E_n))$. Esto es, el alcance de 'λV' se extiende lo más a la derecha posible.
3. $\lambda V_1 \ldots V_n$ representa $(\lambda V_1.(\cdots (\lambda V_n.E) \cdots))$.

93. La aparición de una variable V en una expresiónλ es libre si no aparece en el alcance de λV, de otro modo está ligada. *libre*

3.3 Reglas de conversión de expresiones-λ

94. Las expresiones-λ pueden utilizarse para representar datos como números, cadenas, pares, listas, etc. Cuando se construye una expresión-λ donde se representa cierto cálculo, se puede *convertir* o *reducir* para simplificarla aplicando las reglas conocidas como α, β y η. Estas reglas son muy general y permiten simular la evaluación de expresiones aritméticas, de tratamiento de cadenas, etc.

95. Las reglas de conversión donde se realiza una substitución textual $E[V := E']$ se pueden cometer, solo si son *válidas*, es decir, sii ninguna de las variables libres de E' se convierte (por sorpresa) en una variable ligada en $E[V := E']$.

conversión-α Cualquier abstracción de la forma $\lambda V.\, E$ puede convertirse en $\lambda V'.\, E[V := V']$ si la substitución es válida.

conversión-β Cualquier aplicación de la forma $(\lambda V.\, E_1)\, E_2$ puede convertirse en $E_1[V := E_2]$, si la substitución es válida.

conversión-η Cualquier abstracción de la forma $\lambda V.\, (E\, V)$ puede convertirse en E si V no aparece libre en E.

Se puede representar la conversión de la siguiente forma:

- $E_1 \xrightarrow{\alpha} E_2$ significa que E_1 se α-convierte a E_2.
- $E_1 \xrightarrow{\beta} E_2$ significa que E_1 se β-convierte a E_2.
- $E_1 \xrightarrow{\eta} E_2$ significa que E_1 se η-convierte a E_2.

3.3.1 Conversión-α, o re-denominación de variables

α-redex

96. Una expresión-λ (necesariamente una abstracción) a la que se pueda aplicar una reducción-α se denomina α-redex, donde "redex" representa "expresión reducible". Esta regla viene a decir que las variables ligadas pueden ser renombradas siempre que no haya conflicto de nombres con otras existentes.

Esta regla tiene más que ver con facilidad de realizar la manipulación textual de las cadenas que con otras características más peculiares.

3.3.2 Conversión-β, o evaluación de función

β-redex

97. Una expresión-λ (necesariamente una aplicación) a la que se pueda utilizar una reducción-β se denomina β-redex. La regla de conversión-β es como la evaluación de una llamada a una función en un lenguaje de programación habitual: el cuerpo E_1 de la función $\lambda V.\, E_1$ se evalúa en un entorno en el que el "parámetro formal" V se liga al "parámetro actual" E_2.

Esta regla es la más importante, y es la que se emplea para simular mecanismos de evaluación arbitrarios.

3.3.3 Conversión-η, o conversión a función de primer orden

η-redex

98. Una expresión-λ (necesariamente una abstracción) a la que se pueda utilizar una reducción-η se denomina η-redex. La regla de conversión-η expresa la propiedad de que dos funciones son iguales si proporcionan los mismos resultados cuando se les aplica los mismos argumentos, en el sentido de Leibnitz 1.7.

Esta propiedad es extensional, $\lambda V.\,(E\,V)$ representa la función a la que cuando se le aplica el argumento E' retorna $(E\,V)[V := E']$. Si V no aparece libre en E entonces $(E\,V)[V := E'] = (E\,E')$. Esto es, $\lambda V.\,E\,V$ y E proporcionan ambos el mismo resultado, escrito $E\,E'$, cuando se aplica a los mismos argumentos de donde se deduce que son la misma función.

3.4 Igualdad de expresiones-λ

99. Dos expresiones-λ son iguales si se puede reescribir una en otra por una secuencia de conversiones-λ, tanto en un sentido como en otro. De forma más rigurosa, decimos que dos expresiones-λ E y E' son iguales si son idénticas o si existen expresiones $E_1, E_2, \ldots, E_n$ tales que:

1. E es idéntica a E_1.
2. E' es idéntica a E_n.
3. Para cada i, se cumple una de las dos:
   (a) $E_i \xrightarrow{\alpha} E_{i+1}$ o $E_i \xrightarrow{\beta} E_{i+1}$ o $E_i \xrightarrow{\eta} E_{i+1}$.

(b) $E_{i+1} \xrightarrow{\alpha} E_i$ o $E_{i+1} \xrightarrow{\beta} E_i$ o $E_{i+1} \xrightarrow{\eta} E_i$.

Esta relación de igualdad es reflexiva, simétrica y transitiva, y cumple la ley de Leibnitz, tal y como se describe en la sección 1.3.3

3.5 Funciones importantes en cálculo-λ

100. A pesar de que el cálculo-λ es un lenguaje muy primitivo, se puede emplear para representar la mayoría de objetos y estructuras de la programación moderna, la cuestión es definir funciones cuyo comportamiento tenga las propiedades adecuadas. Así veremos las expresiones booleanas, y los números, así como la forma de conseguir recursividad (e iteratividad).

3.5.1 Expresiones booleanas

101. Hay muchas formas de implantar las expresiones booleanas, la más extendida es la siguiente: Las constantes *cierto*, *falso*, y el operador $\neg$ se representan con las funciones **true**, **false** y **not** que se definen:

(3.1) $\quad \mathbf{true} = \lambda x.\, \lambda y.\, x$

(3.2) $\quad \mathbf{false} = \lambda x.\, \lambda y.\, y$

(3.3) $\quad \mathbf{not} = \lambda t.\, t\ \mathbf{false}\ \mathbf{true}$

Es sencillo demostrar que estas funciones cumplen las propiedades:**not true** = **false** y **not false** = **true**.

102. Las funciones **true** y **false** funcionan también como operadores de selección (condicionales):

$$\mathbf{true}\, E_1\, E_2 = E_1$$
$$\mathbf{false}\, E_1\, E_2 = E_2$$

esto quiere decir, que cualquier función que se evalúe a **true** o **false** funciona como operador de selección.

103. A pesar de lo anterior, resulta útil introducir una notación especial para la función selección que da una notación más manejable:

(3.4) $\quad (E \rightarrow E_1 \,|\, E_2) = (E\, E_1\, E_2)$

donde E es una función como **true** o **false**

104. Apoyándose en lo anterior, se puede construir la función **and** y la función **or**, para implementar $\wedge$ y $\vee$, de la siguiente forma:

(3.5) $\quad \mathbf{and} = \lambda xy.\, x\, y\, \mathbf{false}$

(3.6) $\quad \mathbf{or} = \lambda xy.\, x\, \mathbf{true}\, y$

ó

(3.7) $\quad \mathbf{and} = \lambda xy.\, (x \rightarrow y \,|\, \mathbf{false})$

(3.8) $\quad \mathbf{or} = \lambda xy.\, (x \rightarrow \mathbf{true} \,|\, y)$

3.5.2 Números

105. Igual que en el caso anterior, hay muchas formas de representar números con expresiones-λ y los operadores elementales. Las funciones importantes son **suc** (sucesor), **pre** (predecesor) y **iszero** (¿es cero?). Las propiedades deseadas de estas funciones son:

$$\begin{aligned}
&\mathbf{suc}\,\underline{n} = \underline{n+1}\\
&\mathbf{pre}\,\underline{n} = \underline{n-1}\\
&\mathbf{add}\,\underline{m}\,\underline{n} = \underline{m+n}\\
&\mathbf{iszero}\,\underline{0} = \mathbf{true}\\
&\mathbf{iszero}\,(\mathbf{suc}\,\underline{n}) = \mathbf{false}
\end{aligned}$$

106. Una representación habitual de los números, es la que sigue:

$$\underline{n} = \lambda f x.\, f^n x \tag{3.9}$$

donde, por ejemplo:

$$\begin{aligned}
&\underline{0} = \lambda f x.\, x\\
&\underline{1} = \lambda f x.\, f x\\
&\underline{2} = \lambda f x.\, f(f x) = \lambda f x.\, f^2 x; \ldots
\end{aligned}$$

107. Según la descripción anterior, las operaciones se pueden representar así:

$$\mathbf{suc} = \lambda n f x.\, n\, f\, (f\, x) \tag{3.10}$$

$$\mathbf{add} = \lambda m n f x.\, m\, f\, (n\, f\, x) \tag{3.11}$$

$$\mathbf{iszero} = \lambda n.\, n\, (\lambda x.\, \mathbf{false})\, \mathbf{true} \tag{3.12}$$

La forma de codificar la operación "predecesor" en esta formulación es más compleja. Church, en sus trabajos originales propone una más sencilla.

3.5.3 Recursividad. Punto fijo

108. Para representar la multiplicación, hay que escribir una expresión-λ (**mult**) que realice algo así como:

$$\mathbf{mult}\; m\, n = \underbrace{\mathbf{add}\, n\, (\mathbf{add}\, n\, (\cdots (\mathbf{add}\, n\, \underline{0}) \cdots))}_{m\ \mathbf{add}\mathbf{s}}$$

Podemos escribir una definición de la función **mult** que satisfaga lo anterior:

$$\mathbf{mult} = \lambda m\, n.\, (\mathbf{iszero}\, m \rightarrow \underline{0} | \mathbf{add}\, n\, (\mathbf{mult}\, (\mathbf{pre}\, m)\, n)) \tag{3.13}$$

Sin embargo, no puede usarse para definir **mult** dado que estaríamos definiendo una función consigo misma.

109. Se conoce como *punto fijo* de una función E, aquella función E' tal que $E\,E' = E'$. Se puede demostrar que toda expresión-λ tiene un punto fijo. El camino hacia la recursividad queda abierto si podemos emplear esta técnica para definir funciones arbitrariamente. Para ello debemos diseñar una función cuyo punto fijo pueda ser cualquier función (con una pequeña manipulación).

Una expresión-λ op con la propiedad de que $op\,E = E\,(op\,E)$ para cualquier E se denomina *operador de punto fijo*. Hay infinidad de operadores de punto fijo diferentes, el más conocido es **Y**, definido por: operador de punto fijo

$$\mathbf{Y} = \lambda f.\,(\lambda x.\,f\,(x\,x))\,(\lambda x.\,f\,(x\,x)) \tag{3.14}$$

que exhibe la propiedad $\mathbf{Y}\,E = E\,(\mathbf{Y}\,E)$.

110. Provistos de **Y**, podemos definir **mult**. Para ello empecemos con **multFn**:

$$\mathbf{multFn} = \lambda f\,m\,n.\,(\mathbf{iszero}\,m \to \underline{0}\,|\,\mathbf{add}\,n\,(f\,(\mathbf{pre}\,m)\,n))$$

y definimos

$$\mathbf{mult} = \mathbf{Y}\;\mathbf{multFn}$$

de modo que $\mathbf{mult}\,m\,n = \mathbf{multFn}\;\mathbf{mult}\,m\,n$.

3.6 Resultados importantes del cálculo-λ

111. Se dice que una expresión E está "totalmente evaluada" si no es posible aplicar ninguna reducción β o η, es decir la única transformación posible es renombrar variables. Entonces se dice que la expresión está en *forma normal*. forma normal

3.6.1 Teorema de Church-Rosser

112. "Si $E_1 = E_2$ entonces existe una E tal que $E_1 \longrightarrow E$ y $E_2 \longrightarrow E$". En otras palabras viene a decir que si se evalúa una expresión E de dos formas diferentes hasta que se obtienen dos formas normales E_1 y E_2, cada una es α-convertible a la otra. El corolario del teorema de Church-Rosser proporciona interesantes conclusiones sobre las formas normales:

1. Si E tiene una forma normal, entonces $E \longrightarrow E'$ para algún E' en forma normal.

2. Si E tiene una forma normal y $E = E'$, entonces E' también tiene una forma normal.

3. Si $E = E'$ y E y E' están ambas en forma normal, entonces E y E' son idénticas salvo α-conversión.

3.6.2 Teorema de normalización.

113. Puede ocurrir que al reducir una expresión-λ, algunos caminos conduzcan a una secuencia infinita de conversiones (p.ej.: $(\lambda x.1)\,(\mathbf{Y}\,f)$, pero otros caminos conduzcan a una forma normal. El teorema de normalización permite evitar estos callejones sin salida si la expresión tiene forma normal (detalle que no siempre se puede conocer de antemano). El teorema dice: "Si E tiene una forma normal, si se aplica repetidamente la reducción de la β o η-redex del lado izquierdo, el proceso terminará, y se obtendrá una expresión en forma normal".

orden normal

114. Esta secuencia de reducción se denomina "de *orden normal*" y no tiene por que ser la secuencia más eficiente computacionalmente. En el ámbito de la programación, esta forma de evaluación se denomina *evaluación por nombre*, diferida, perezosa, o "*lazy*", y por regla general exige del intérprete más recursos que la evaluación denominada *evaluación por valor*, ávida o "*eager*".

evaluación por nombre

evaluación por valor

3.7 Implementaciones de calculadoras-λ

115. La programación mediante lenguajes de programación que implementan alguna versión del cálculo-λ se conoce como programación funcional. Aunque el paradigma de programación no es nuevo, y existen versiones desde antiguo (LISP), solo en la actualidad se valora adecuadamente sus ventajas. La utilización de estos lenguajes se ve motivada por dos razones fundamentales: aprendizaje de la tecnología de la programación y aplicación en áreas donde la programación imperativa falla o resulta muy onerosa. En este ultimo área se encuentran desarrollos como: manipulación de información simbólica (theorem proving, model checking, natural language,...), construcción de compiladores, prototipado rápido, verificación formal de programas.

116. Los lenguajes que implementan el cálculo-λ presentan construcciones sintácticas reformadas ("sintactic sugar") para facilitar la escritura y comprensión de programas. Sus características peculiares, como la transparencia referencial (con ausencia de "variables" al estilo habitual), el enfoque recursivo de la programación y la facilidad de implementación de abstracciones (funcionales y de tipo de datos) lo hacen muy útil para realizar aplicaciones y prototipado para manipulación de información en un nivel de abstracción elevado. Al mismo tiempo suelen ser la mesa de trabajo de lenguajes en proceso de evolución constante donde se plantean las alternativas más innovadoras al diseño de compiladores y lenguajes de programación desde un punto de vista rigurosamente formal.

117. Actualmente se puede clasificar los lenguajes de programación funcional según dos criterios: modo de evaluación y tipado de datos.

En cuanto al modo de evaluación, SML, HOPE, LISP, Scheme proveen un esquema de llamada por valor, con lo que la evaluación de los argumentos que se pasan a cada función se evalúan antes de efectuar la función. Haskell, Miranda, LML, etc emplean evaluación diferida, que es más rigurosa pero menos eficiente en algunos casos.

Lenguajes como Scheme, FP y LISP, proporcionan herramientas para la gestión de tipos de datos, denominándose estos lenguajes "no tipados". La interpretación del tipo de datos empleado en cada función depende del uso realizado del dato, de modo que los errores en el uso de los tipos de datos se notifican en tiempo de ejecución (si se advierten). Esto proporciona flexibilidad, pero se pierde la rigurosidad de la teoría de tipos y se carece de las herramientas de alto nivel de gestión de tipos de datos como las que incorpora Haskell, Miranda, CaML y ML.

4

Autómatas

Índice General

4.1 Introducción

4.1.1 Autómata como abstracción de procesamiento de cadenas de símbolos.

118. Un automatismo es un dispositivo que opera autónomamente sin la intervención de la mano del hombre, realizando algún proceso. Este proceso puede ser de índole físico, como un robot, una planta química o un avión, o de índole simbólico, como el proceso de gestión económica de una empresa.

En informática, un autómata es un dispositivo que manipula cadenas de símbolos que se le presentan a su entrada produciendo otras tiras o cadenas de símbolos como salida. Este dispositivo existe como ente abstracto y en sí mismo, la informática lo estudia como un modelo de procesamiento de información. Este estudio permitirá diseñar sistemas físicos o lógicos que se comporten obedeciendo al modelo especificado, sea este un ordenador de propósito general o un control de gobierno de un automatismo. También permite modelar sistemas existentes y estudiarlos desde esta perspectiva, como descripción del funcionamiento de seres vivos, sistemas socioeconómicos, etc.

119. Un ordenador es un ejemplo de autómata, y también lo son componentes menores de este sistema, como contadores, decodificadores, sumadores, etc. Desde esta perspectiva, se utilizarán a cada nivel de estudio de autómata, unos modelos y unas herramientas diferentes.

4.1.2 Máquinas secuenciales y maquinas combinacionales. Concepto de estado

120. El estudio de los sistemas de procesamiento de información distingue entre los procesos que obedecen de modo unívoco para cada entrada en cada momento (procesos combinatorios o combinacionales), y los sistemas que obedecen a un modelo en el que hay que tener en cuenta la serie de entradas anteriores para considerar cada salida posible (procesos secuenciales).

121. Es habitual que en las máquinas secuenciales para una misma entrada instantánea la salida pueda ser diferente si se considera un instante diferente. Esto es así porque su comportamiento hay que considerarlo estudiando cada secuencia de entradas. Para entender esto se recurre a la idea de que una máquina secuencial presenta "configuraciones internas instantáneas", responsables de que se produzcan diferentes salidas con la misma entrada. A esta abstracción la denominamos *estado*. Como es lógico, el conjunto de estados y la ley de comportamiento puede conocerse por diseño del autómata o mediante la inspección más precisa posible del comportamiento de un autómata ya existente. Podemos construir diferentes modelos de estados para una misma máquina y presentar todos el mismo comportamiento externo.

estado

122. La ley de comportamiento de un autómata no solo incluye el símbolo que emite en cada estado, frente a cada uno de los posibles símbolos de entrada, sino la

ley de transición que indica cual será el estado al que pasa dicha máquina.

4.1.3 Autómatas y lenguajes

123. El estudio de los autómatas y de los lenguajes está muy unido. Partiendo de las definiciones de alfabeto, y reglas gramaticales de un lenguaje, cualquier cadena de símbolos que pertenezca a este lenguaje puede ser objeto de reconocimiento por lo que se denomina *autómata reconocedor* de este lenguaje. Este proceso de reconocimiento puede ser utilizado para generar acciones o traducir estas acciones a otro lenguaje, siendo en cada caso, para cada tarea y para cada lenguaje, un tipo de autómata peculiar.

autómata reconocedor

4.1.4 Autómatas y álgebra

124. La formación de las sentencias de un lenguaje puede formalizarse desde el punto de vista algebraico. La operación es la concatenación, y el conjunto de cadenas con concatenación tiene estructura algebraica de semigrupo, al que si se añade un elemento neutro se convierte en monoide. En función del tipo de autómata reconocedor (si tiene un conjunto de estados finito), se puede establecer un número finito de clases de equivalencia en el lenguaje de entrada, lo que permite definir un semigrupo (o monoide) cociente, llamado semigrupo (o monoide) de la máquina de cuyas propiedades puede formalizarse cuestiones de interés sobre autómatas.

4.2 Autómatas finitos

4.2.1 Definición y representación de autómatas. Autómatas de Moore y de Mealy.

125. Un autómata es un *sistema de transiciones etiquetádas*, y se define formalmente mediante una quíntupla: $A = \langle E, S, Q, f, g\rangle$, donde:

E *alfabeto* de entrada que contiene los símbolos de entrada.

S conjunto finito de *salida*, que contiene los símbolos de salida.

Q conjunto de *estados*.

$f : E \times Q \to Q$, función de *transición* o evolución..

$g : E \times Q \to S$, función de *salida*.

Esta máquina al recibir una entrada $e \in E$ en el instante t y estando en el estado $q \in Q$, da una salida $g(e, q)$, y pasa al estado $f(e, q)$ en el instante $t + 1$ –si expresamos el tiempo para expresar de modo genérico el compás de evolución del sistema.

Si A tiene un conjunto Q finito el autómata es finito (AF).

126. Las funciones f y g pueden representarse mediante una tabla con tantas filas como estados y tantas columnas como entradas, en la casilla (j, i) aparece la función de transición y la función de salida $f(e_j, q_i)/g(e_j, q_i)$ del autómata para la entrada e_j y el estado q_i.

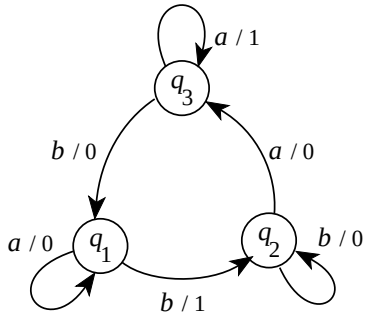


Figura 4.1: Ejemplo de diagrama de transiciones.

diagrama de transiciones

Otra forma de representarlo es como un grafo dirigido, en el que cada nodo corresponde a un estado q_i y de cada estado parte una flecha por cada símbolo de entrada posible, donde se especifica qué símbolo de salida se produce y cuál es el estado siguiente. Este esquema se denomina *diagrama de transiciones* o diagrama de Moore.

máquina de Mealy

máquina de Moore

127. El autómata en el que para las transiciones a un mismo estado podemos tener diferentes salidas se clasifica como autómata o *máquina de Mealy* y su representación es como la que se ha visto anteriormente. El autómata en el que para todas las transiciones a un mismo estado se tiene la misma salida, se clasifica como autómata o *máquina de Moore* y presenta algunas peculiaridades interesantes. Todo autómata de Mealy tiene una representación de Moore que presenta el mismo comportamiento, aunque posiblemente con un número de estados mayor.

monoide libre

semigrupo libre

128. En el autómata de Moore se puede considerar un elemento más en el alfabeto de entrada, el elemento neutro ϵ, de modo que se amplía el alfabeto a $\{E\} \cup \{\epsilon\}$. Este elemento equivale físicamente a la ausencia de entrada. En cuanto a la función de transición, se entiende que la ausencia de entrada no modifica el estado. En cuanto a la función de salida, se entiende que se produce la salida asociada al estado. Esta posibilidad es propia de los autómatas de Moore puesto que en éstos, la salida es función, únicamente, del estado. El lenguaje universal asociado a una máquina de Moore es un *monoide libre* $\langle E^* \rangle$, mientras que en la máquina de Mealy solo es un *semigrupo libre* $\langle E^+ \rangle$.

4.2.2 Comportamiento entrada-salida de los autómatas

129. La formulación de autómata finito de la sección anterior, no es la única. Una formulación alternativa especialmente útil desde el punto de vista del lenguaje y equivalente a la anterior es la siguiente. Un autómata finito es una quíntupla $\langle Q, \Sigma, \delta, q_0, F\rangle$, donde Q es un conjunto finito de estados, Σ es un alfabeto finito de entrada, q_0 en Q es el estado inicial, $F \subset Q$ es el conjunto de estados finales, y δ es la función de transición $Q \times \Sigma$ sobre Q, de modo que $\delta(q, a)$ es un estado para cada estado q y símbolo de entrada a.

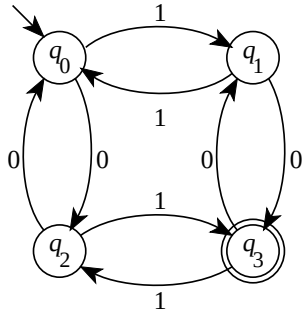


Figura 4.2: Diagrama de transición de un autómata reconocedor.

Como se puede apreciar, no aparecen símbolos de salida, se parte de la idea de que si el autómata ofrece diversas acciones, estas se representan como el hecho de llegar a un cierto estado del conjunto de estados finales. Desde este punto de vista, se puede pensar que el autómata responde con un cierto símbolo de salida que representa un estado intermedio, y tantos otros símbolos de salida como estados finales. Esta formulación está encaminada a facilitar el trabajo con cadenas de símbolos de entrada que ponen en funcionamiento el autómata, y como parece razonable, el comienzo de una cadena de entrada se sincroniza con un autómata en su estado inicial.

130. Para describir formalmente el comportamiento de un autómata finito, sobre una cadena, debemos extender la función δ para aplicarla a un estado y a una cadena. Se define la función $\hat{\delta}$ del $Q \times \Sigma^*$ hacia Q. $\hat{\delta}(q, w)$ es el estado del autómata finito tras leer w comenzando en el estado q, es decir, el único estado p tal que hay un camino en el diagrama de transición de q a p, etiquetado con w.

1. $\hat{\delta}(q, \epsilon) = q$, y
2. para cada w y símbolo de entrada a, $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$.

131. Se dice que una cadena x es *aceptada* por un autómata finito $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ si $\hat{\delta}(q_0, x) = p$ para algún p en F. El *lenguaje aceptado* por M, y se designa $L(M)$, es el conjunto $\{x|\hat{\delta}(q_0, x) \in F\}$. Un lenguaje es un *conjunto regular* si es el conjunto aceptado por algún autómata finito.

lenguaje aceptado

conjunto regular

4.2.3 Autómata finito no determinista

132. El concepto de no determinismo es importante en teoría de lenguajes y en la teoría de la computación. Se puede concebir un autómata finito no determinista, si se permite cero, una o más transiciones de estado con el mismo símbolo de entrada. Se puede demostrar que el conjunto aceptado por un autómata finito no determinista puede ser aceptado por un autómata finito determinista, aunque ésta clase de autómatas sea un caso de la clase más general –no determinista.

En este tipo de autómatas, la noción de estado deja de ser el concepto intuitivo que aparece en las máquinas deterministas. Aunque un autómata no determinista no pueda estar en varios estados a la vez, a la vista de lo expuesto hasta ahora, no parece claro como podemos saber en cuál de ellos se encuentra un autómata en todo momento.

133. Dado que desde un estado hay más de una posible transición con un símbolo de entrada dado, para determinar si una cadena es aceptada por un autómata de este tipo es necesario hallar un camino que conduzca a un estado final o aceptor. Al obtener la secuencia de estados posible, se obtiene un árbol donde se desglosa cada alternativa posible. De este modo δ ahora es una aplicación de $Q \times \Sigma$ a 2^Q. Ahora $\delta(q, a)$ es el conjunto de todos los estados p tales que hay una transición de q a p etiquetada a.

134. De igual modo que anteriormente, se puede extender δ a $\hat{\delta}$ aplicando $Q \times \Sigma^*$ sobre 2^Q:

1. $\hat{\delta}(q, \epsilon) = \{q\}$.
2. $\hat{\delta}(q, wa) = \{p|$para algún estado r en $\hat{\delta}(q, w)$, p está en $\delta(r, a)\}$.
3. $\hat{\hat{\delta}}(P, w) = \bigcup_{q \in P} \hat{\delta}(q, w)$. Con $P \subset Q$.

Autómatas finitos con ϵ-movimientos

135. Es posible extender el autómata finito no determinista para incluir transiciones con la entrada vacía ϵ. En consecuencia este tipo de autómatas permite aceptar una cadena w, partiendo de un estado q, sin tener en cuenta nuevos símbolos de entrada. En este caso, tiene especial interés averiguar qué vértices son alcanzables desde cada estado q –ϵ-CIERRE(q)– pues la noción de aceptabilidad ha de incluir como estados finales q, si el estado aceptor pertenece al cierre.

4.3 Expresiones regulares

136. Los lenguajes aceptados por los autómatas finitos se describen fácilmente mediante expresiones denominadas expresiones regulares. Se puede demostrar que la clase de lenguajes aceptados por los autómatas finitos es precisamente la clase del lenguajes descriptibles por las expresiones regulares.

137. Dado un alfabeto Σ y dos conjuntos de cadenas L_1 y L_2 formado por símbolos de este alfabeto podemos definir la concatenación de L_1 y L_2 (L_1L_2) como $\{xy|x \in L_1 \wedge y \in L_2\}$. Si se define $L^0 = \{\epsilon\}$ y $L^i = LL^{i-1}$ para $i \leq 1$ el *cierre de Kleene* es el conjunto de todas las cadenas que se pueden formar a partir de L, y se escribe L^*. cierre de Kleene

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

.

138. Las expresiones regulares sobre Σ y los conjuntos que denotan se definen recursivamente como sigue:

1. $\emptyset$ es una expresión regular y denota el conjunto vacío.
2. ϵ es expresión regular y denota el conjunto $\{\epsilon\}$.
3. Para cada a en Σ, **a** es expresión regular y denota el conjunto $\{a\}$.
4. Si r y s son expresiones regulares que denotan R y S, respectivamente,
   (a) $(r + s)$ es expresión regular y denota $R \cup S$.
   (b) (rs) es expresión regular y denota RS.
   (c) (r^*) es expresión regular y denota R^*.

4.3.1 Aplicaciones de los autómatas finitos

139. Existe una variedad de problemas en el diseño del software que se pueden simplificar si se dispone de generadores de autómatas finitos eficientes, a partir de la expresión regular que identifica el autómata. De este modo, se escribe la expresión regular, y cierta herramienta genera el código fuente correspondiente al autómata finito, o bien se encarga de interpretar la entrada del problema obedeciendo al comportamiento especificado.

140. Un ejemplo habitual es el trabajo con las palabras y operadores de los lenguajes de programación ("tokens"). Por poner un ejemplo, en ALGOL los identificadores de variables, funciones, tipos, ... comienzan con una letra (**A** + **B** + $\cdots$ + **Z** + **a** + **b** + $\cdots$ + **z**) y se sigue con una serie de letras y/o dígitos (**0** + **1** + $\cdots$ + **9**). Esto se expresa en la expresión regular:

$$(\text{letra})(\text{letra} + \text{dígito})^*$$

En FORTRAN, con identificadores limitados a seis caracteres:

$$(\text{letra})(\epsilon + \text{letra} + \text{dígito})^5$$

141. En ciertos editores de texto y similares, se permite la búsqueda y substitución de una cadena atendiendo a un criterio de búsqueda por expresiones regulares (patrón de búsqueda). En algunos casos incluso es posible realizar búsqueda aproximada de cadenas, proporcionando una distancia que exhibe la proximidad al patrón de búsqueda del patrón encontrado.

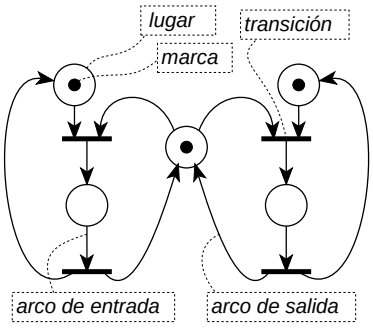
4.4 Otros autómatas

142. Si bien el autómata finito revisado anteriormente es un modelo capaz de dar cuenta del comportamiento de multitud de sistemas discretos, con dinámica temporal y memoria, a menudo, es necesario contemplar otros tipos de autómatas más adecuados para expresar otras situaciones como: un número de estados infinito, transiciones probabilistas, pertenencia difusa a conjuntos, y autómatas con aprendizaje. En la práctica, estos autómatas son una herramienta que proporciona un nivel de abstracción más adecuado al problema, aunque pueda simularse su comportamiento con autómatas finitos.

4.4.1 Redes de Petri

143. Estos autómatas son una herramienta muy útil para modelar sistemas en los que se dan actividades asíncronas y concurrentes, desde el diseño de programas concurrentes y con comunicación hasta el estudio de sistemas de producción y economía.

144.



Una de las representaciones más habituales de las Redes de Petri es mediante grafos, pero con la diferencia importante de que en estos, los nodos de los grafos no representan estados de la Red. En el grafo de una Red de Petri pura existen dos tipos de nodos: círculos para los *lugares* (capaces de almacenar *marcas*), segmentos para las *transiciones* y arcos de dos tipos *de incidencia anterior* o *de entrada* (desde los lugares a las transiciones) y *de incidencia posterior* o *de salida* (desde las transiciones a los lugares). Las marcas se representan como puntos en el interior de los lugares.

145. En cada configuración inicial de una Red de Petri, hay una asignación de marcas a los lugares. Si todos los lugares que preceden a una transición están marcados, se quita una marca a cada uno de ellos y se pone una marca a cada lugar que esté precedido por esa transición. Una característica importante es que en cada evolución de una red de Petri, pueden dispararse varias transiciones simultáneamente (otra diferencia con los grafos en los autómatas finitos). El *estado* de la red es la asignación de marcas en cada instante. La teoría de Redes de Petri permite averiguar propiedades generales de los sistemas partiendo de su modelo de Red de Petri asociado y las propiedades del modelo de marcado del sistema concreto. propiedades

146. Para aplicar el concepto de Red de Petri al modelar sistemas se establece una correspondencia entre las entradas del sistema con las condiciones necesarias de disparo y las salidas al disparo de otras transiciones o a las marcas en ciertos lugares.

4.4.2 Autómatas estocásticos

147. Un autómata estocástico o probabilístico es una quíntupla $AP = \langle E, S, Q, P, h\rangle$ donde E, S y Q tienen el significado habitual, h es la función de salida (en el sentido de autómata de Moore) y P es un conjunto de probabilidades, asociadas a cada una de las transiciones. La suma de las probabilidades de ir a cualquier estado, partiendo de un estado, dado un símbolo de entrada, es 1. Esto viene a decir que, partiendo de cierto estado y con cierto símbolo de entrada tenemos cierta probabilidad de ir a cada uno de los estados del autómata.

148. Para cada símbolo de entrada a tenemos una matriz de transición $M(a) = \{p_a(i,j)\}$ donde $p_a(i,j)$ es la probabilidad de que el estado siguiente al estado q_i sea el estado q_j.

149. Este tipo de autómatas se emplea en sistemas en los que aparece cierto grado de incertidumbre y aparecen variables aleatorias. Estos, se deben modelar desde un punto de vista estadístico, como por ejemplo experimentos de tiradas al azar y cualquier sistema modelables con procesos markovianos. Un ejemplo típico son los sistemas de comunicación, y en especial los asociados al proceso de comunicación oral y el lenguaje natural.

En estos casos, lo habitual es que planteado un modelo razonable, el diseñador se vea en la necesidad de obtener las probabilidades de transición mediante procedimientos iterativos de entrenamiento.

4.4.3 Autómatas con aprendizaje

150. La noción de autómata con aprendizaje, proviene principalmente del campo de la inteligencia artificial, y se puede encuadrar dentro de los sistemas reactivos con adaptación al ambiente. La idea básica es que el autómata presenta un comportamiento frente al entorno que puede verse modificado por la acción de un esquema de adaptación al entorno, que evalúa los cambios del entorno y las prestaciones globales del sistema. La política de estas modificaciones persiguen siempre el mejor desempeño del sistema que es en el fondo una medida de prestaciones junto con un criterio de optimalidad.

151. El punto de vista de los autómatas con aprendizaje, o adaptativos, suele emplearse en el caso de autómatas estadísticos, donde es posible aproximar las probabilidades de transición del autómata en torno a un punto óptimo, sin modificar las demás características del autómata. En otros contextos la noción de autómata suele referirse a otro tipo de sistemas con una estructura interna más compleja que la estudiada en este apartado (agentes).

4.4.4 Autómatas borrosos

152. La definición de autómata borroso es muy similar a la presentada para el autómata estocástico, salvo que la función de transición cobra otra interpretación

diferente. En este caso, la función de transición borrosa presenta dos diferencias importantes:

1. los coeficientes no son probabilidades, luego no es necesario que exista normalización para cada estado de partida y símbolo de entrada.

2. El cálculo de la probabilidad de estados para una cadena de símbolos de entrada es puntual. Mientras que en el caso de las probabilidades, la probabilidad del estado final es una probabilidad conjunta, en este caso se calcula teniendo en cuenta únicamente el estado más favorable.

153. Este tipo de representación tiene utilidad en el modelado de sistemas orientados al control y en reconocimiento de patrones, donde la naturaleza de los estados y de las transiciones resulte difícil de establecer con precisión.

5

Conceptos de computabilidad. Máquina de Turing

Índice General

154. De modo general, entendemos por *computabilidad* la factibilidad de programar una computadora para resolver cierto problema. Este problema se puede plantear en términos de que planteado un hipotético algoritmo que pretende realizar un cómputo, ¿es este algoritmo efectivo o factible?. Una de las propiedades de todo algoritmo es su *efectividad*, es decir, se puede llevar a cabo con una computadora, si bien no se restringe la cantidad de recursos empleados, siempre que no sean infinitos.

computabilidad

efectividad

155. Para responder a la pregunta de qué es computable, se han construido modelos de cómputo que si bien son de carácter abstracto conservan la misma capacidad

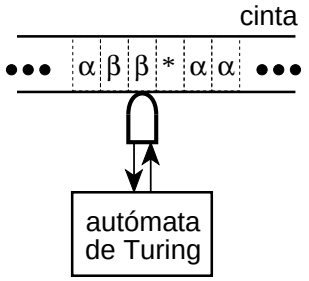
de cómputo que las computadoras usuales. Sobre estos modelos, entre los que se halla la máquina de Turing (MT) se formulan los términos de computabilidad.

Teorema de Incompletitud

En 1931 K. Gödel demostró el *Teorema de Incompletitud* que asevera que no existe un procedimiento general para el cálculo de de muchas funciones. Ello dio al traste con las intenciones de D. Hilbert (principios del siglo XX) de encontrar un algoritmo que determinara la verdad o falsedad de cualquier proposición matemática.

5.1 Máquina de Turing

5.1.1 Definición y funcionamiento de MT.



156. Una máquina de Turing es un autómata finito, junto con una cinta de longitud infinita (que en cualquier momento contiene solo un número finito de símbolos) que se encuentra dividida en casillas (cada casilla puede contener un solo símbolo o estar en blanco) y un aparato para explorar (leer) una casilla, imprimir un nuevo símbolo sobre ella. Esta máquina posiblemente se desplace sobre la cinta una casilla a la derecha o a la izquierda hasta que posiblemente llegue a un estado en que se detenga (stop) indefinidamente. Existen formulaciones ligeramente diferentes a ésta aunque equivalentes, en las que la cinta está limitada por la izquierda.

157. Para resolver un problema concreto hay que diseñar una MT específica $\mathcal{M} = \langle \Sigma, \Sigma \times M, Q \cup \{\text{stop}\}, f, g \rangle$ donde:

Σ alfabeto externo de los símbolos de la cinta. Incluye el símbolo vacío $\circ$ para las casillas en blanco.

M conjunto finito de movimientos de la cadena. En este caso $\{\leftarrow, \rightarrow, \leftrightarrow\}$: retroceder, avanzar y no moverse, respectivamente.

Q alfabeto interno con los estados del autómata de control.

$f : \Sigma \times Q \rightarrow Q \cup \{\text{stop}\}$ función de transición del autómata de control.

$g : \Sigma \times Q \rightarrow \Sigma \times M$ función de salida. Par con el símbolo nuevo de la cinta y el movimiento.

stop es un estado del autómata que simplifica la escritura, en lugar de incluir un estado aceptor común y corriente.

Además hay que especificar:

- Información inicial del problema, codificada en una palabra A formada por símbolos del alfabeto externo y que se halla escrita en la cinta.

- Estado inicial del autómata de control.

158. Partiendo de que la cabeza se halla sobre el símbolo no vacío que está más a la izquierda, la máquina se pone a funcionar del siguiente modo:

1. Lectura del símbolo de la cinta bajo la cabeza.
2. Obtención del nuevo estado del autómata, el nuevo símbolo a escribir en la cinta y el movimiento a realizar con la cabeza. Tras lo cual cambia el estado del autómata de control y se escribe el símbolo nuevo.
3. Se lleva a cabo el movimiento.

159. Cada MT define un lenguaje $L(\mathcal{M})$, de modo que cualquier palabra $A \in L(\mathcal{M})$ que se escriba en la cinta, ocasionará que la máquina se pare en un número finito de pasos, con la cabeza sobre el comienzo de la palabra de salida. Sin embargo para palabras no pertenecientes a este lenguaje, es posible que la máquina se detenga o no.

160. En el siguiente ejemplo se propone una MT que suma dos números naturales que se codifican en unario con $\bullet$ y se separan con el símbolo $\diamond$. El alfabeto externo Σ es $\{\circ, \bullet, \diamond\}$ y se emplean tres estados $\{q_0, q_1, q_2\}$. El carácter que está bajo la cabeza de lectura se indica con un acento circunflejo. La descripción de la MT es como sigue:

	Símbolo de entrada		
Estado	$\circ$	$\bullet$	$\diamond$
q_0	$(\circ, q_0, \rightarrow)$	$(\circ, q_2, \rightarrow)$	$(\circ, \mathsf{stop}, \leftrightarrow)$
q_1	$(\circ, q_0, \rightarrow)$	$(\bullet, q_1, \leftarrow)$	$(\diamond, q_1, \leftarrow)$
q_2	$(\bullet, q_1, \leftrightarrow)$	$(\bullet, q_2, \rightarrow)$	$(\diamond, q_2, \rightarrow)$

Para el caso "$2 + 3$" ($\bullet\ \bullet \diamond \bullet\ \bullet\ \bullet$) la secuencia de *descripciones instantáneas* es como sigue:

$q_0, \circ\ \hat{\bullet}\ \bullet \diamond \bullet\ \bullet\ \bullet \circ\ \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \hat{\bullet} \diamond \bullet\ \bullet\ \bullet \circ\ \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \bullet\ \hat{\diamond}\ \bullet\ \bullet\ \bullet \circ\ \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \bullet \diamond \hat{\bullet}\ \bullet\ \bullet \circ\ \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \bullet \diamond \bullet\ \hat{\bullet}\ \bullet \circ\ \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \bullet \diamond \bullet\ \bullet\ \hat{\bullet} \circ\ \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \bullet \diamond \bullet\ \bullet\ \bullet\ \hat{\circ}\ \circ\ \circ$
$\vdash\ q_1, \circ\ \circ\ \bullet \diamond \bullet\ \bullet\ \bullet\ \hat{\bullet} \circ\ \circ$
$\vdash\ q_1, \circ\ \circ\ \bullet \diamond \bullet\ \bullet\ \hat{\bullet}\ \bullet \circ\ \circ$
$\vdash\ q_1, \circ\ \circ\ \bullet \diamond \bullet\ \hat{\bullet}\ \bullet\ \bullet \circ\ \circ$
$\vdash\ q_1, \circ\ \circ\ \bullet \diamond \hat{\bullet}\ \bullet\ \bullet\ \bullet \circ\ \circ$

$\vdash\ q_1, \circ\ \circ\ \bullet\ \hat{\diamond}\ \bullet\ \bullet\ \bullet\ \bullet \circ\ \circ$
$\vdash\ q_1, \circ\ \circ\ \hat{\bullet} \diamond \bullet\ \bullet\ \bullet\ \bullet \circ\ \circ$
$\vdash\ q_1, \circ\ \hat{\circ}\ \bullet \diamond \bullet\ \bullet\ \bullet\ \bullet \circ\ \circ$
$\vdash\ q_0, \circ\ \circ\ \hat{\bullet} \diamond \bullet\ \bullet\ \bullet\ \bullet \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \circ\ \hat{\diamond}\ \bullet\ \bullet\ \bullet\ \bullet \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \circ\ \diamond\ \hat{\bullet}\ \bullet\ \bullet\ \bullet \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \circ\ \diamond \bullet\ \hat{\bullet}\ \bullet\ \bullet \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \circ\ \diamond \bullet\ \bullet\ \hat{\bullet}\ \bullet \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \circ\ \diamond \bullet\ \bullet\ \bullet\ \hat{\bullet} \circ\ \circ$
$\vdash\ q_2, \circ\ \circ\ \circ\ \diamond \bullet\ \bullet\ \bullet\ \bullet\ \hat{\circ}\ \circ$
$\vdash\ q_1, \circ\ \circ\ \circ\ \diamond \bullet\ \bullet\ \bullet\ \bullet\ \hat{\bullet} \circ$

$\vdash\ q_1, \circ\ \circ\ \circ\ \diamond \bullet\ \bullet\ \bullet\ \hat{\bullet}\ \bullet \circ$
$\vdash\ q_1, \circ\ \circ\ \circ\ \diamond \bullet\ \bullet\ \hat{\bullet}\ \bullet\ \bullet \circ$
$\vdash\ q_1, \circ\ \circ\ \circ\ \diamond \bullet\ \hat{\bullet}\ \bullet\ \bullet\ \bullet \circ$
$\vdash\ q_1, \circ\ \circ\ \circ\ \diamond \hat{\bullet}\ \bullet\ \bullet\ \bullet\ \bullet \circ$
$\vdash\ q_1, \circ\ \circ\ \circ\ \hat{\diamond}\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet \circ$
$\vdash\ q_1, \circ\ \circ\ \hat{\circ}\ \diamond\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet \circ$
$\vdash\ q_0, \circ\ \circ\ \circ\ \hat{\diamond}\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet \circ$
$\vdash\ \mathsf{stop}, \circ\ \circ\ \circ\ \circ\ \hat{\bullet}\ \bullet\ \bullet\ \bullet\ \bullet \circ$

El estado q_0 se emplea para iniciar el trabajo y detectar el final de la tarea, el estado q_2 para acarrear unidades desde la parte izquierda a la parte derecha

y el estado q_1 para retroceder hasta la parte izquierda. En otras máquinas de Turing para otras tareas diferentes, el número de estado puede llegar a ser mucho mayor, y el alfabeto externo contener muchos más símbolos, algunos de los cuales se presentan durante el funcionamiento de la máquina para representar descripciones instantáneas de estados más complejos.

5.1.2 Diseño de MT.

161. El diseño de máquinas de Turing, describiendo el conjunto de estados completo y la función de movimiento, es una tarea muy complicada. Para atajar este problema se proponen herramientas conceptuales que están próximas a las técnicas de programación habituales.

Almacenamiento en el autómata – El elemento de control puede utilizarse para almacenar una cantidad finita de información. Para implementarlo, el estado se escribe como un par de elementos, uno que ejerce el control habitual y el otro almacenando el símbolo. En la práctica, esto supone un desdoblamiento en tantos estados equivalentes, desde el punto de vista del control del autómata, como símbolos queremos almacenar, con la única diferencia de que en cada uno se tiene una referencia implícita a un símbolo concreto.

Pistas múltiples – Se puede imaginar que la cinta de la máquina de Turing se divide en k pistas, para un k finito. Los símbolos se consideran k-tuplas, con una componente por pista.

Símbolos de comprobación – Consiste en insertar símbolos que reflejan resultados intermedios, como la comprobación de operandos, y operaciones ya realizadas.

Desplazamientos – La MT puede hacer espacio en la cinta desplazando símbolos un número finito de celdas hacia la derecha o izquierda.

Subrutinas – Como con los programas habituales, el diseño "modular" o "descendente" se facilita si se emplean subrutinas para definir procesos elementales. Una máquina de Turing puede simular cualquier tipo de subrutina encontrada en un lenguaje de programación, incluyendo procedimientos recursivos y cualquier tipo de mecanismo de paso de parámetros conocido.

Para llevar esto a cabo se diseñan MT con un estado inicial y un estado final en el que se realiza el retorno de la llamada. Se especifican estados privados a la subrutina y funciones de transición y movimiento. La llamada se efectúa con una transición a un estado inicial y la vuelta con una transición al estado de retorno. Al incorporarlo en otra máquina de Turing, es preciso hacer una re-denominación de estados y de símbolos si hubiera algún tipo de colisión, esto si, manualmente.

Diagramas Las posibles representaciones de autómatas son extensibles al diseño de máquinas de Turing, entre las que se encuentra la representación habitual de Mealy, con sus modificaciones, y la representación en diagramas de flujo

similares a los empleados en "programación estructurada" para representar los detalles generales.

5.1.3 Simulación de MT.

162. La simulación de una máquina de Turing es una tarea especialmente indicada para llevarse a cabo en una computadora de propósito general, por su carácter rutinario y sistemático. Los programas de simulación disponibles permiten un nivel de trabajo similar al de los ensambladores de alto nivel conocidos.
163. Todas las máquinas de Turing pueden ser simuladas por otra MT llamada Máquina de Turing Universal (MTU), siempre que se le proporcione la descripción completa de la primera. Los dos puntos a tener en cuenta en esta descripción son:

- Cualquier MT puede describirse empleando el alfabeto binario mediante paquetes de 1's y 0's, de este modo la MTU puede disponer de cualquier alfabeto que no entre en colisión con los símbolos anteriores.
- Es deseable que la MT a simular trabaje con una cinta limitada por un extremo (lo cual no resta generalidad a la MT), aunque no sea una condición necesaria. Bajo esta premisa, la MTU puede hacer uso de la parte izquierda libremente, sin que interfiera la MT simulada.

Bajo estas premisas en la cinta que utilizará la MTU estará descrita, por orden, el conjunto de estados, una codificación de las funciones de transición y movimiento de la MT y la palabra que describe el problema a resolver. La peculiaridad de la MTU consiste en que nada impide simular una MTU con otra MTU, o incluso otras MT más complejas. Bajo esta perspectiva, resulta interesante ver cómo una máquina puede desarrollar procesos más complejos que los que su propia estructura parece permitirle, a condición de que se le suministre la información adecuada y los recursos espacio-temporales necesarios.

5.1.4 Variantes de la MT

Existen variantes de máquinas de Turing, por ejemplo:

- MT con solo dos de las tres salidas posibles (símbolo de salida, movimiento, estado siguiente).
- MT con cinta limitada por un extremo.
- MT con más de una cinta y más de una cabeza.
- MT no determinista.

Es posible demostrar que ninguna restringe las posibilidades de la MT vista anteriormente, siendo que cada una de ellas presenta ventajas formales para casos concretos.

5.2 Función computable

5.2.1 Hipótesis de Church-Turing

164. La hipótesis de Church (o Turing) dice que la noción intuitiva informal de un procedimiento efectivo sobre secuencias de símbolos es idéntica a la de nuestro concepto preciso de un procedimiento que puede ser ejecutado por una máquina de Turing. Aunque no existe prueba formal de esta hipótesis, siempre que ha sido intuitivamente evidente, hasta la fecha, que existía un algoritmo para calcular una función recursiva, ha sido posible diseñar una MT que la ejecuta.

5.2.2 La MT como un calculador de funciones enteras.

165. La máquina de Turing puede verse como un calculador de funciones de enteros a enteros, no hay más que representar cada argumento en unario separándolos con ceros $1^{i_1}01^{i_2}0\cdots01^{i_k}$. Una función se denomina parcial, si no está definida para todas las posibles tuplas $(i_1 \dots i_k)$. De otro modo se dice que es total o que está totalmente definida.

función computable

166. Una *función f es computable* si existe una máquina de Turing $\mathcal{M}$ tal que $f(i_1 \dots i_k) = \mathcal{M}(i_1 \dots i_k)$ en el dominio de f, donde partiendo de la misma tupla la máquina para y se obtiene el valor adecuado. El comportamiento de $\mathcal{M}$ fuera del dominio no está definido.

167. La Máquina de Turing es un buen mecanismo para construir funciones computables parciales. Es decir es buena recorriendo conjuntos recursivamente numerables y generando elementos consecutivamente partiendo de los números enteros. El problema principal es que se puede demostrar (Gödel) que existen conjuntos recursivamente numerables para los cuales no es posible obtener una función computable que diga si cierto elemento pertenece al conjunto o no.

La consecuencia directa es que hay ciertos problemas planteados en forma de la teoría de números (como quien dice cálculo de predicados), para los que no es posible obtener una función que compute el resultado (análogamente, que derive todo teorema de los axiomas).

5.2.3 Numerabilidad de las máquinas de Turing.

168. El marco de la teoría de funciones recursivas es muy útil hablando de máquinas de Turing. De hecho, cualquier MT se puede describir mediante un número entero. Si cada MT se describe por una quíntupla, cuyos componentes son conjuntos finitos, ergo numerables, de alguna forma podemos atribuir un número natural a cada una de las quíntuplas. Esto significa que las MT pueden ordenarse numéricamente, aunque el mecanismo de numeración pueda ser algo complejo. Una codificación posible es mediante números de Gödel.

El interés de poder numerar las máquinas de Turing está en poder aplicar los conocimientos que se poseen sobre el sistema axiomático de los números enteros para poder obtener resultados fundamentales sobre la computabilidad.

169. La codificación de Gödel es muy simple y se basa en que cada número tiene una única descomposición como producto de potencias de números primos.

$$p_0^{y_0} \times p_1^{y_1} \times \cdots \times p_n^{y_n}$$

siendo p_i números primos. De esta forma, si coleccionamos todos los símbolos de nuestro alfabeto y se numeran consecutivamente (o no), podemos asignar consecutivamente cada número primo a cada una de las posiciones de los símbolos de una expresión, sea esta aritmética, lógica, textual, etc, y el código de cada símbolo funciona como el exponente del número primo correspondiente. Cada cadena del lenguaje tiene un número, y números diferentes corresponden a cadenas diferentes. El conjunto es numerable.

170. El procedimiento es sistemático, y podemos construir MT que transcriban descripciones de máquinas de Turing en sus correspondientes números de Gödel. También podemos construir MT que dado un número de Gödel diga si es transcribible como una máquina de Turing, y en este caso obtener su transcripción.

5.2.4 Problema de la aplicabilidad y de la parada.

171. El problema de la aplicabilidad equivale a averiguar si es posible construir una MT $\mathcal{M}_1$ que tomando como argumento una descripción de otra MT $\mathcal{M}_2$ y cierta palabra w formada por símbolos del alfabeto de entrada de $\mathcal{M}_2$, w pertenece al lenguaje aceptado por esta última. Este problema se resuelve propiamente en el terreno de la teoría los conjuntos recursivos y recursivamente numerables; clases de conjuntos de números naturales.

172. Trasladando los resultados del Teorema de Incompletitud, a la computablidad, ya se ha dicho que no siempre es posible construir una función computable que permita saber la solución a cierto problema finito aunque seamos capaces de generar las soluciones de todo el dominio del problema, de forma recursiva.

En otras palabras, podemos construir una MT que genere todas las MT conocidas. E incluso podemos construir una que genere todas las MT que sean capaces de averiguar si otra MT acepta la cadena w. Pero no es posible construir una que dado *cualquier* $\mathcal{M}_2$ y w sea capaz de saber si esta cadena será aceptada por la máquina indicada.

173. El *problema de la parada* consiste en si es posible construir una MT $\mathcal{M}_1$ que sea capaz de saber si otra máquina $\mathcal{M}_2$ parará cuando se le suminstra cualquier cadena w. El resultado anterior también se aplica aquí, de modo que no es posible construir una máquina de Turing que sea capaz de decidir si *cualquier* máquina de Turing parará en algún momento para alguna entrada de un problema.

problema indecidible

Se dice que un problema es indecidible si no somos capaces de proporcionar un procedimiento que nos diga si es posible resolver el problema. El problema de la aplicabilidad y el problema de la parada son indecidibles.

6

Complejidad

Índice General

174. Partiendo de una solución algorítmica de un problema, la teoría de la complejidad proporciona herramientas y criterios para clasificar la eficiencia de la solución propuesta. En función del tamaño del problema y de la disponibilidad de recursos, resolver ciertos problemas puede ser inviable a pesar de tener un algoritmo efectivo y bien conocido, aunque no eficaz.

6.1 Complejidad y máquinas de Turing

175. La complejidad no es una medida de la dificultad conceptual de un algorítmico, sino de la inversión en recursos necesaria para aplicar el algoritmo a los datos iniciales, y se enmarca teóricamente a través de las máquinas de Turing, aunque es posible estudiarla sobre algoritmos realizados en lenguajes de programación habituales. Uno de los resultados más interesantes es que dicha medida de complejidad

no depende una máquina concreta sobre la que se ejecute, aunque sí demuestra influencia la paralelización sobre varios procesadores.

6.1.1 Criterios de complejidad: espacial y temporal.

176. La condición de parada sobre la máquina de Turing (MT) equivalente es el único requerimiento de efectividad para un algoritmo. En él no tienen cabida límites de espacio en la cinta ni número de pasos de ejecución. Sin embargo, se puede medir la eficacia al resolver un problema, mediante varias MT equivalentes. Algunas requerirán mas longitud de cinta que otras, otras mayor número de pasos, otras requieren varias cintas, otras cintas limitadas por un extremo, etc.

177. Una máquina de Turing TM tiene una *complejidad espacial* $S(n)$ si, para cualquier palabra de entrada de longitud n, explora como máximo $S(n)$ celdas de la cinta diferentes a las celdas de la palabra de entrada (celdas de trabajo). También se dice que TM está $S(n)$-limitada en espacio.

178. Una máquina de Turing TM tiene una *complejidad temporal* $T(n)$ si para cualquier palabra de entrada de longitud n, TM realiza como máximo $T(n)$ movimientos antes de detenerse. También se dice que TM está $T(n)$-limitada en tiempo.

179. Al plantear la complejidad, en algunas ocasiones se emplea la expresión completa de $S(n)$ y $T(n)$ y en otras se emplean términos asintóticos, todo depende de la magnitud de la tarea o del orden de comparación entre dos algoritmos.

6.1.2 Máquinas deterministas y no deterministas.

180. En las máquinas deterministas para una misma palabra de entrada, la solución se alcanza, si es el caso, en un número concreto de etapas, empleando un conjunto concreto de celdas, independientemente del momento o las veces que se ponga en funcionamiento. En las máquinas no deterministas, no siempre se invierte el mismo número de etapas y de celdas cada vez que se pone en funcionamiento, pues la traza de instrucciones realizadas depende de decisiones tomadas en el momento de la ejecución y de las que no es responsable el algoritmo, dado que este solo establece las posibilidades.

Un ejemplo de esto lo tenemos en una búsqueda en un vector de datos: puede ser determinista (secuencial) o puede ser no determinista (arbitraria). De este modo, las definiciones de complejidad son ligeramente diferentes.

181. Una máquina de Turing no determinista es $S(n)$-limitada en espacio (o de complejidad espacial $S(n)$) si, para toda entrada de longitud n, la máquina no debe indagar más de $S(n)$ posiciones de la cinta, cualquiera que sea la secuencia de decisiones que pueda tomar durante la ejecución del algoritmo.

182. Una máquina de Turing no determinista es $T(n)$-limitada en tiempo (o de complejidad temporal $T(n)$) si, para toda entrada de longitud n, la cabeza de lectura de la máquina efectúa menos de $T(n)$ desplazamientos, cualquiera que sea la secuencia de decisiones.

6.1.3 Algunos resultados sobre MT.

183. Las complejidades espacial y temporal tienen unos límites inferiores. Si cualquier máquina de Turing tiene al menos una celda, entonces $S(n) \geq 1$ para cualquier n, es decir cuando se habla de complejidad espacial se emplea la expresión $\max(1, \lceil S(n) \rceil)$.

De igual modo para la complejidad temporal, si al menos hay que leer $n+1$ símbolos, si el problema de la cinta mide n, $T(n) \geq n+1$, de modo que al igual que más arriba, la complejidad temporal resulta ser $\max(n+1, \lceil T(n) \rceil)$.

184. Una propiedad interesante de la complejidad, es que no resulta afectada por constantes sobre n. Si una máquina TM $S(n)$-limitada en espacio y de k cintas acepta cierto lenguaje L, también existen máquinas $cS(n)$-limitadas en espacio que aceptan L, siempre que $c > 0$.

Este resultado es comprensible, dado que variando la codificación del dato de entrada de la MT podemos comprimir o expandir arbitrariamente la longitud de la representación en un factor c. Ello conlleva una complicación o simplificación de la MT correspondiente.

185. Otro resultado interesante, permite asegurar que si una máquina TM $S(n)$-limitada en espacio y de k cintas acepta cierto lenguaje L, existen máquinas $S(n)$-limitadas en espacio que también aceptan L. Lo que nos permite decir que $S(n)$ establece clases de equivalencia entre las máquinas de Turing, no importa el número de cintas.

186. Si cierta máquina de Turing $T(n)$-limitada en tiempo y de k cintas acepta cierto lenguaje L, con $k > 1$ y $\inf_{n\to\infty} T(n)/n = \infty$, existen máquinas $cT(n)$-limitadas en tiempo que aceptan L, siempre que $c > 0$.[1]

187. Mientras que la complejidad espacial no aumenta en el caso de una o varias cintas, no ocurre lo mismo en el caso de la complejidad temporal, siendo que en el peor de los casos, al pasar de varias a una cinta que la complejidad temporal se convierte en $T(n) \cdot T(n)$. De modo que es de esperar un cambio cualitativo en la complejidad de la solución de un problema mediante un proceso secuencial o mediante procesos concurrentes.

Si cierta MT $T(n)$-limitada en tiempo y de k cintas acepta cierto lenguaje L, con $k > 1$, cualquier máquina de una cinta que acepta este lenguaje está $T^2(n)$-limitada en tiempo.

En el caso de pasar de una MT con $k > 2$ a $k = 2$, la limitación temporal pasa de $T(n)$ a $T(n) \cdot \log(T(n))$

6.2 Complejidad algorítmica

188. Cuando se resuelve un problema, con frecuencia hay que elegir entre varios algoritmos. En esta búsqueda, hay dos objetivos que suelen contra decirse:

[1] La expresión $\inf_{n\to\infty} f(n)$ es el límite cuando $n \to \infty$ de la mayor de las cotas inferiores de $f(n), f(n+1), \ldots$.

que el algoritmo sea fácil de entender, codificar y depurar, y que el algoritmo use eficientemente los recursos del computador y, en especial, que se ejecute con la mayor rapidez posible. Del estudio de la complejidad teórica de cada algoritmo y de mediciones reales de tiempos de ejecución se obtiene el algoritmo idóneo.

Si el programa se usa pocas veces, el primer objetivo es el más importante, pero en ciertas ocasiones es necesario elegir aquél más eficiente. Entre estas se encuentra:

- Programas en tiempo real, y programas con tiempo de retorno limitado.
- Programas con volumen de almacenamiento limitado físicamente.
- Programas de utilización frecuente, que pueden degradar las prestaciones del sistema

189. Para muchos programa, el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ella. En este caso se suele emplear el tiempo de ejecución del *peor caso*. Para hacer referencia a la velocidad de crecimiento de los valores de una función, se usará la notación conocida como *notación asintótica* (O grande). Por ejemplo, decir que el tiempo de ejecución $T(n)$ de un programa es $O(n^2)$, que se lee "o grande de n al cuadrado" o tan solo "o de n al cuadrado", significa que existen constantes enteras positivas c y n_0 tales que para $n \geq n_0$, se tiene $T(n) \leq cn^2$.

Se dice que $T(n)$ es $O(f(n))$ si existen constantes positivas c y n_o tales que $T(n) \leq cf(n)$ cuando $n \geq n_0$. Cuando el tiempo de ejecución de un programa es $O(f(n))$, se dice que tiene *velocidad de crecimiento* $f(n)$. Entonces $f(n)$ es una cota superior para la velocidad de crecimiento de $T(n)$. Para especificar una cota inferior para la velocidad de crecimiento de $T(n)$, usa la notación $T(n)$ es $\Omega(g(n))$, que se lee "$T(n)$ es omega grande de $g(n)$" o simplemente "$T(n)$ es omega de $g(n)$", lo cual significa que existe una constante c tal que $T(n) \geq cg(n)$ para un número infinito de valores de n.

6.2.1 Complejidad polinómica y exponencial

190. Los algoritmos cuyo comportamiento asintótico es del tipo $O(n)$, $O(n^2)$, $O(n^3)$, ..., en general $O(n^c)$ para c constante, se llaman *algoritmos polinómicos*, o de *complejidad polinómica*. Entre algoritmos de la misma clase, con diferente c son preferibles aquellos que tienen un índice menor, siendo en algunos casos necesario realizar un estudio empírico de la implementación. Los algoritmos con comportamiento asintótico como 2^n, o en general c^n, son algoritmos exponenciales, o de *complejidad exponencial*. Este tipo de algoritmos se convierte en intratable incluso con valores pequeños de n. Suele emparejarse con algoritmos de búsqueda exhaustiva de soluciones.

La figura **??** permite visualizar la variación del orden de magnitud de cuatro algoritmos de complejidad (que supondremos temporal): 2^n, n^2, $n \log n$ y n.

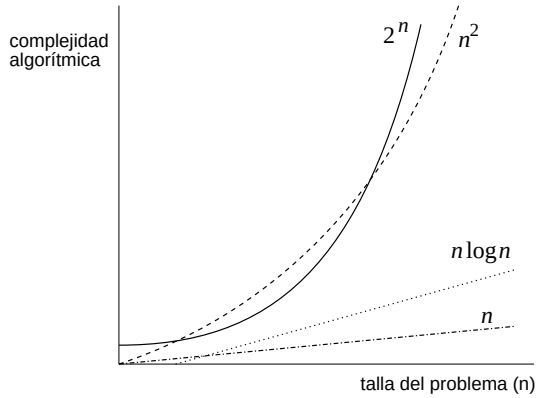


Figura 6.1: Diversos órdenes de complejidad.

6.2.2 Ejemplos de algoritmos con complejidad polinómica.

191. Los ejemplos siguientes resultan de los problemas de búsqueda y clasificación, para los que se proporcionan algoritmos de un uso muy frecuente y bien estudiado. El problema de búsqueda se reduce a la selección de un elemento de un conjunto al que se puede acceder a través de un campo clave. En este caso se entiende que el coste de acceso al elemento, una vez dada su clave, es 1.

192. Con el algoritmo de *búsqueda secuencial* se explora consecutivamente, y obedeciendo a una secuencia fija de claves, todos los elementos del conjunto. Para cada elemento, desde el primero, se realiza una comprobación para averiguar si es el elemento que nos interesa. En el peor de los casos, el coste no es mayor que el número de elementos del conjunto, n. Tanto la complejidad temporal como la espacial son $O(n)$.

193. En la búsqueda binaria, se parte de la premisa de que la posición de los elementos respetan un orden, conforme a la secuencia de claves. El algoritmo en sí va acotando la posición del elemento en cuestión, tomando el conjunto total y accediendo al elemento central. Este algoritmo tiene complejidad $O(log_2 n)$. Si en cada etapa de acceso y comparación, el tamaño del problema se reduce a la mitad, el número máximo de comparaciones que pueda necesitar el método de búsqueda será de $log_2(n)$.

194. El problema de la clasificación consiste en organizar el conjunto de objetos en secuencia, de modo que el acceso ordenado a las claves de acceso a los elementos proporcione un acceso ordenado a los elementos, según cierta relación de orden.

En el método de ordenación de la burbuja, se recorre cierto numero de veces el conjunto, intercambiando la posición de los elementos consecutivos que no respeten la relación de orden. Es de esperar que si esto se hace un número suficiente de veces, al final el conjunto esté ordenado. No es tema de este punto tratar el algoritmo

en concreto, aunque sí, podemos decir, que en el mejor de los casos, el número de comparaciones es $\frac{1}{2}n(n-1)$, y el número de intercambios (en el peor de los casos) es $\frac{3}{2}(n^2-n)$. De este modo, la complejidad es del orden de $O(n^2)$.

195. En el método de ordenamiento "quick-sort" de C.A.R. Hoare, la complejidad es del orden de $O(n \cdot \log n)$. Este algoritmo emplea una aproximación similar a la búsqueda binaria, en cuanto a que el tamaño del problema se reduce en cada iteración.

6.3 Problemas P, NP y NP-completos

196. Dado que los problemas más duros son aquellos que tienen una complejidad de exponencial, por contraposición se entiende que los problemas abordables son aquellos para los que podemos encontrar un algoritmo *determinista* con complejidad polinómica, aunque el exponente del coeficiente no sea muy pequeño, esta clase de problemas se denomina clase P. Una de las metas de la programación consiste en encontrar algoritmos eficientes para los problemas, en la realidad, existen problemas para los que no se puede encontrar una solución determinista que se evalúe en tiempo polinómico.

6.3.1 El problema P-NP.

197. Sin embargo, para ciertos problemas para los que no se conoce un algoritmo de clase P si se conocen algoritmos *no deterministas* que lo resuelven en un tiempo polinómico. Esta clase de problemas se denominan de clase NP (por No-deterministas Polinómicos, no confundir con No Polinómicos). Un ejemplo habitual es encontrar un circuito Hamiltoniano en un grafo, para el que no existe una solución determinista polinómica, pero el algoritmo de sortear ciclos y verificar si son un ciclo de Hamilton si es polinómico.

La diferencia entre P y NP es análoga a la diferencia entre encontrar eficientemente una demostración de una sentencia, y el de verificar eficientemente tal demostración de la sentencia.

198. Uno de los problemas centrales en la teoría de la complejidad es averiguar si un problema pertenece a una de estas dos clases. Lo que está claro es que para cualquier problema P existe un algoritmo de tipo NP, es decir: la clase de problemas P está incluida en NP. Pero *nadie sabe si NP incluye problemas que no estén en P*. El tema de si NP está propiamente contenido en P está abierto.

6.3.2 Completitud y problemas NP-completos

199. Una de las posibilidades de solución es estudiar aquellos problemas NP que parezcan más difíciles (duros) y demostrar que no son P. Una categoría de problemas que presenta ciertas propiedades de dificultad se denomina problemas NP-completos. Para esta clase, un resultado sorprendente demuestra que si se encon-

trara una solución P para uno cualquiera de ellos, la clase completa de problemas NP estaría contenida en P.

200. Un problema paradigmático, entre todos, es el problema de la satisfacibilidad, o *problema SAT*. Este problema se plantea cuando buscamos una interpretación que dé el valor cierto al evaluar con ella una expresión booleana. Los problemas de satisfacibilidad en otras lógicas son derivados de este problema, y no añaden dificultad. También se ha demostrado formalmente que todos los problemas NP-completos conocidos hasta el momento pueden reducirse al problema SAT.

problema SAT

201. Existe una gran variedad de problemas NP-completos, entre los cuales enumeramos:

- El problema de la satisfacibilidad.
- El problema del recorrido Hamiltoniano.
- El problema de del número cromático (o del coloreado de mapas).
- El problema del agente de ventas (o del viajante).
- El problema de las particiones.

Hasta la fecha no se conoce la equivalencia de P y NP, por lo que se supone razonablemente que P y NP no son iguales, como hipótesis de trabajo en la teoría de la complejidad.