El significado de computar

*Juan C. Hidalgo-Del Vecchio**

Introducción

A finales del siglo pasado y principios del corriente el matemático alemán David Hilbert (1862-1943) se embarcó en la tarea de encontrar un algoritmo discreto para determinar la falsedad o veracidad de cualquier proposición matemática. Lo que Hilbert buscaba en particular era un proceso mecánico para determinar la veracidad de una fórmula arbitraria en Cálculo de Predicados de Primer Orden aplicada a números enteros.

La meta era (y es) muy ambiciosa; todos nos hemos enfrentado en una u otra ocasión con problemas matemáticos por lo que querer obtener una receta para determinar veracidad, aplicable para cualquier proposición, es de interés evidente. Para poder utilizar métodos matemáticos es necesario tener notación apropiada con la cual describir los conceptos involucrados. El Cálculo de Predicados de Primer Orden (CPPO) es una notación (o "lenguaje") para describir relaciones entre funciones y conjuntos. Muchas proposiciones matemáticas pueden ser descritas con gran sencillez, en forma compacta y (¡sobre todo!) ***sin ambigüedad.*** Hilbert pensaba que el modelo de predicados de primer orden era tan poderoso que toda la matemática podía ser expresada en éste; tristemente no es posible describir totalmente conceptos tan simples como los números enteros. Sin embargo, el CPPO se ha convertido en una herramienta de mucha utilidad en la Matemática y Computación (justamente por las razones expresadas anteriormente); su dominio es un requerimiento imperativo de cualquier estudiante actual en ambas ciencias.

La Matemática no es una disciplina que se preocupa intrínsecamente por el tiempo que toma efectuar u proceso o los recursos requeridos por éste. Hilbert m esperaba que el algoritmo pudiera determinar si un lema matemático era falso o verdadero "rápidamente' La aplicación del algoritmo podía durar, pero por li menos debía terminar eventualmente; esto en sí implicaba que el algoritmo tenía que poder describirse e espacio finito (aunque nótese que un algoritmo de tamaño finito puede aplicarse indefinidamente, como en el caso de *ciclos o "loops").* Metafóricamente, Hilbert buscaba el catálogo de la Biblioteca de Babel borgiana.

En 1931 Kurt Gódel (1906-) publicó su famoso *Teorema de Incompletitud* el cual demostraba (entre otras cosas) cómo el algoritmo buscado por Hilbert n podía existir. Su prueba consistía en construir una formula en CPPO aplicada a enteros cuyo significado (sea, su misma definición) acertaba que ésta no podía ser probada ni negada dentro de CPPO. La formalización de este argumento, así como la subsecuente clarificación y formalización de nuestra idea intuitiva d *computares* uno de los grandes triunfos intelectual del siglo XX.

Una vez formalizada la noción de *Procedimiento Efectivo* (PE - procedimiento que siempre termina) se pudo demostrar que tampoco existen PE para muchos otros problemas matemáticos, entre ellos el poder contar los elementos de grandes colecciones de funciones. Consideremos el conjunto **B** de funciones que mapean los números enteros no negativos al conjunto {0, 1}. Es fácil construir un isomorfismo (una corre pondencia 1-1) entre el conjunto **B** y los numere reales. Por otro lado, dado que los PE tienen que si descritos en forma finita, es también posible construir

* Juan Carlos Hidalgo Del Vecchio, Ph. D. Profesor en la Escuela Ciencias de la Computación e Informática Universidad de Costa Rica. e-mail: juanh@cariari.ucr.ac.cr

un isomorfismo entre el conjunto de procesos efectivos y los números naturales. En resumen, tenemos que hay tantos elementos en **B** como números reales y además que hay tantos PE como números enteros, lo que implica que hay más funciones en **B** que procesos efectivos. Esto significa que existen muchas funciones para las que no es posible describir un algoritmo (o programa) que las compute. Estas funciones son denominadas ***no computables.*** Curiosamente, el hecho de que existan funciones no computables no es tan sorprendente como el que existan problemas y funciones en esta clase que cuentan con gran relevancia en la Matemática, Ciencias de la Computación y otras disciplinas.

La formalización de Procedimiento Efectivo más usada en la actualidad es la Máquina de Turing (MT), aunque (como veremos) no es la única forma de definir el concepto. Lo verdaderamente interesante es que todos los modelos de cómputo propuestos hasta la fecha, incluyendo las Máquinas de Turing, han resultado ser equivalentes entre ellos. En particular, la MT es equivalente en poder de cómputo a la computadora digital que conocemos en la actualidad y a las nociones matemáticas de cómputo más generales. Como es de suponer, no es posible demostrar que el modelo de Turing corresponde a nuestra idea intuitiva de cómputo, pero existen argumentos de peso en esa dirección; estos se conocen como la Tesis de Church.

El Concepto de Algoritmo

La humanidad ha utilizado máquinas para resolver problemas por miles de años. El género introducido en tiempos modernos es la máquina que ejecuta una colección de direcciones externas para efectuar sus tareas. Las máquinas anteriores eran construidas de tal forma que solo podían efectuar secuencias específicas de instrucciones internas para resolver un problema en particular (pues solo podían efectuar las operaciones para las que habían sido diseñadas). Hoy construimos máquinas que cuentan con un gran repertorio de instrucciones y que pueden ser programadas para múltiples tareas; la gran diferencia es la "memoria" de la computadora moderna. La memoria permite almacenar secuencias de códigos que, al ser interpretadas por la máquina, corresponden a secuencias de instrucciones. La memoria es también importante por el tipo de instrucciones que permite definir, en particular las ***instrucciones condicionales.*** La memoria nos permite escribir programas para instruir a la computadora a hacer casi cualquier cosa.

Existen tres conceptos alrededor de la noción de "cómputo":

- **Plan de acción:** un esquema general de los pasos principales necesarios para obtener una meta a partir de un punto inicial predeterminado; normalmente un plan de acción es descrito en lenguaje natural.
- **Algoritmo:** refinamiento de un plan de acción. En un algoritmo solo podemos utilizar acciones con semántica clara de tal forma que no haya campo para ambigüedad o incertidumbre. Todas las escogencias son hechas con criterios explícitos y en forma determinística.
- **Programa:** es la implementación de un algoritmo en un lenguaje de programación.

Como podemos ver, la noción de algoritmo es un concepto intuitivo por lo que su formalización fue el objetivo seminal de la Teoría de Computabilidad. Esta disciplina se interesa en el estudio de modelos para expresar algoritmos y caracterizaciones matemáticas de funciones computables. La teoría quedó cimentada en la década de 1930 cuando se propusieron varias definiciones matemáticas para la *noción de función computable por algoritmo.* Se demostró matemáticamente que toda función sobre los enteros no negativos es computable si y solo si es posible definirla bajo ciertos esquemas y/o ecuaciones. Luego se argumentó convincentemente, aunque sin prueba formal, que toda función algorítmica es computable por una Máquina de Turing (la noción matemática actual de "algoritmo"). Por último, se demostró que esta tesis, de ser cierta, trae implicaciones: ciertos problemas teóricos y prácticos son "indecidibles", o dicho de otra forma, no existen algoritmos para resolverlos (toda "solución" falla para algún valor de entrada). Esta excitante contribución de la Lógica Matemática de esos años se ha convertido en uno de los pilares de la Computación teórica actual.

Los algoritmos han sido de importancia para las matemáticas desde la antigüedad; se acepta como primer algoritmo conocido al procedimiento para calcular el máximo común divisor de dos enteros descrito por Euclides en *Los Elementos.* No obstante, la palabra "algoritmo" fue utilizada por primera vez en tiempos medievales. El término nació a partir del nombre del gran matemático Islámico Abü Ja'far Muhammad ibn Musa al Kharizmi que murió alrededor de 847 p.c. Muchos trabajos históricos se refieren a este hombre como "Al-Kharizmi", un epíteto que normalmente indica que era nativo de Kharizm ahora la cuidad de Khiva y sus alrededores, 100 kms. al sur del Mar Aral en el antiguo Uzbekistán soviético. Se sabe que vivió por un período corto en Kharizm aunque residió durante la mayor parte de su vida en Bagdad (ahora capital de Iraq), la capital del Imperio Islámico y el gran centro académico del momento.

Los pensadores Islámicos entre el siglo Vil y XIII fueron autores de grandes contribuciones en el desa-

rrollo de la matemática. El trabajo principal de Al-Kharizmi fue el libro de texto sobre matemáticas elementales titulado "El Libro Completo sobre Cálculo bajo Complemento y Balance" (trabajo parcialmente original y también resultado de la transmisión de importantes ideas matemáticas griegas e hindúes). Contenía un sistema completo de reglas para manipulación algebraica además de material sobre el sistema numérico hindú; la referencia al término "al-jabr" en*el título original del libro es la fuente de la palabra "álgebra". Una vez traducido al latín en el siglo XII, el texto fue tan influyente en Europa que para los siglos XIII y XIV toda la aritmética con el sistema numérico hindú (en vez de numerales romanos) era identificada con At-Kharizmi; la forma latina de su nombre "algorismus" fue utilizada en todos los tratados de álgebra. La palabra "algoritmo" y su significado actual son producto de dicho uso.

Modelos de Cómputo

La caracterización de Turing

Todo paradigma formal para modelaje de procedimientos efectivos debe poseer ciertas características fundamentales. Primero, debe ser posible describir cualquier procedimiento dentro del modelo en forma finita. Segundo, todo procedimiento debe consistir de pasos discretos que puedan ser ejecutados en forma mecánica. Un modelo con dichas características fue propuesto en 1936 por Alan Turing (1912-54). Las Máquinas de Turing consisten de tres componentes básicos: un elemento de estados finitos como control, una cinta y una cabeza para leer/escribir sobre la cinta. La cabeza actúa como línea de comunicación entre la unidad de control y la cinta al leer y escribir símbolos sobre ella. La unidad de control opera en forma discreta; la máquina ejecuta una de las siguientes operaciones dependiendo del estado actual y el símbolo que está siendo reconocido por la cabeza en la cinta:

1. La unidad de control cambia del estado actual a otro estado sin mover la cabeza.
2. Se escribe un símbolo sobre el espacio de la cinta que está siendo leído o se mueve la cabeza un espacio para la izquierda/derecha.

La cinta tiene un borde izquierdo, pero se extiende indefinidamente hacía la derecha. Como la máquina puede mover la cabeza un espacio a la vez, después de todo cómputo finito sólo se visitará una cantidad finita de cinta. La máquina cesa de operar si se trata de mover la cabeza a la izquierda del extremo (izquierdo) de la cinta.

Inicialmente la cinta de la máquina es utilizada para suministrar valores de entrada a la MT (inscribiendo la información de izquierda a derecha sobre ésta);

el resto de la cinta consiste de espacios en blanco. La máquina es libre de modificar los valores de entrada, así como de escribir indefinidamente sobre la porción en blanco de la cinta.

Dado que una MT puede escribir sobre toda su cinta, los resultados del cómputo pueden dejarse como impresiones sobre ésta. Toda MT cuenta con un estado de término para señalar el final del cómputo, aunque no es posible inferir de este hecho ninguna connotación, favorable o desfavorable, sobre el proceso efectuado. Para Turing el sentido formal de "computar" consiste en la secuencia de acciones que efectúa una MT a partir del estado inicial hasta que la máquina se detiene (si es que llega a detenerse). En resumen, las Máquinas Turing fueron diseñadas para satisfacer los siguientes criterios:

1. Deben ser autómatas, por lo que su construcción y funcionalidad puede ser estudiada con los mismos parámetros utilizados para dispositivos mecánicos.
2. Deben ser lo más simple posible de definir formalmente y de tal forma que se pueda razonar sobre ellas.
3. Deben ser tan generales como sea posible (en términos de los cómputos que pueden efectuar).

Las Máquinas Turing no son una clase de autómatas que pueden ser reemplazadas por otro modelo de cómputo más poderoso. Aunque de aspecto primitivo, todo intento de aumentar su poder de cómputo (agregar más cintas o cabezas) transforma a una MT en otra MT con las mismas capacidades de cómputo. Estos resultados son demostrados vía simulación, técnica que consiste en convertir una MT "aumentada" en una MT estándar que funciona en forma análoga. De este modo todo cómputo que puede ser efectuado por una MT especial puede ser efectuado por una MT ordinaria.

Como hemos visto, las MT forman una clase maxi-mal de autómatas en términos de los cómputos que pueden efectuar. Más adelante veremos que las MT son equivalentes en poder de cómputo a otros modelos totalmente distintos. Aún de más importancia, veremos que todo intento (hasta el momento) de formalizar el concepto de "procedimiento computacional" y/o "algoritmo" ha resultado ser equivalente a la noción de Máquina Turing.

La Caracterización Kleene

Al principio de la década de los 30 Kurt Gódel (1906-) y Stephen Kleene (1909-) concentraron su trabajo en formular una definición para el concepto de

función recursiva general. Kleene presentó su definición en 1936 y él mismo da los créditos a Gódel. Gódel creía que la recursividad general debía ser considerada como criterio para la noción de "procedimiento efectivo" (el término utilizado por Gódel era "procedimiento efectivamente calculable"). La presentación de Kleene, en su estructura general, es la siguiente.

Uno de los métodos para caracterizar clases de funciones consiste en definir una colección de ecuaciones cuyas soluciones constituyen los miembros de la clase que se intenta definir; si las ecuaciones utilizadas son recursivas se dice que la clase cuenta con una *definición recursiva.* Una definición recursiva para funciones es, en forma general, una descripción donde el valor de la función asociado a sus argumentos depende del valor de la misma función sobre argumentos más "simples" o a valores de funciones más "simples". La noción de "simple" es parte de la especificación de la definición; generalmente se utilizan funciones constantes, entre otras, como las funciones más simples.

Kleene define a **C** (la clase de *funciones primitivas recursivas)* como el conjunto de funciones más pequeño tal que:

1. Todas las funciones constantes *(η_k = k* son elementos de C
2. La función sucesor s(n) = n + 1 es elemento de **C**
3. Las funciones de proyección π_k (n_1, …, n_k) = n_i
   son elementos de **C**
4. El conjunto **C** es cerrado bajo *composición,* lo que significa que si en **C** existe *g* una función de I parámetros y h_1, h_l son funciones de k parámetros, entonces la función *f(n) = g(h₁(n),* ... , *h*,(n)) donde n € N^k, también es un elemento de **C**
5. El conjunto C es cerrado bajo *recursión primitiva,* lo que significa que si en C existe *g* una función de k parámetros y *h* es una función de k+2 parámetros, entonces la única función *f* de k+1 parámetros que satisface las siguientes ecuaciones

$$f(n,O) = g(n)$$

$$f(n,m + 1) = n(n,m,f(n,m))$$

donde n € N^k, también pertenece a **C**.

Lo que interesaba a Kleene era determinar la extensión del conjunto **C**; ¿es **C** lo suficientemente general para incluir todos los algoritmos deseados? y, en consecuencia, ¿una forma apropiada para definir formalmente la noción informal de *función computable por algoritmo?* Aunque las reglas de definición para funciones primitivas recursivas parecen limitadas a primera vista, es posible destacar una gran cantidad de evidencia para sostener dichas hipótesis. Por ejemplo, se puede demostrar que casi toda función matemática ordinaria es primitiva recursiva. Sin embargo, no toda función que naturalmente aparenta ser computable es primitiva recursiva. Se puede obtener un ejemplo con una simple aplicación del *Principio de Diagonalización,* argumento muy utilizado en la computación teórica. Dado que toda función primitiva recursiva corresponde a la aplicación de una de las cinco reglas mencionadas, es fácil listar sistemáticamente a todas las funciones de C. Esto significa que podemos crear mecánicamente una lista infinita A_1, A_2, A_3, donde cada elemento de la lista representa la definición de una función primitiva recursiva; además que toda función primitiva recursiva aparece como algún elemento de la lista. Definimos a f_i como la función primitiva recursiva A_i y consideramos la función *f* que toma un entero positivo n y computa $f(n) = f_n(n) + 1$. *f* es claramente computable dado que para calcular el resultado lo que debe hacer es listar las funciones recursivas hasta llegar al número n, evaluar la definición de A_n con n como argumento y sumar 1 al resultado de la evaluación. Sin embargo, *f* no puede ser primitiva recursiva; supongamos que lo fuera, esto implicaría que *f* pertenece a la lista de funciones primitivas recursivas como A_i para algún i. Pero si evaluamos a *f* en i tendríamos que $f(i) = f(i) + 1$, lo cual es claramente imposible. Este ejemplo, aunque simple, no provee un ejemplo de funciones "naturales" que, aunque computables, no son primitivas recursivas. Existen ejemplos de funciones claramente algorítmicas (como la conocida función *Exponencial Generalizado de Ackermann)* que no son primitivas recursivas, sin embargo, los argumentos involucrados en las demostraciones de no pertenencia a **C** son bastante elaborados.

En vista de las limitaciones de la clase de funciones primitivas recursivas, Kleene definió el concepto de *minimalización sin restricción;* ésta fue la expansión que necesitaban las funciones primitivas recursivas para coincidir con la clase de funciones computables. Para Kleene, un conjunto de instrucciones **P** consiste en un conjunto de "ecuaciones recursivas". El proceso de *cómputo* corresponde a una secuencia finita (o infinita) de ecuaciones donde cada ecuación en la lista se obtiene a partir de:

- Una ecuación anterior en la lista con substituciones uniformes en las variables.
- Sustitución de "iguales por iguales".
- Evaluación de instancias de la función sucesor.

La caracterización de Church

El Cálculo Lambda (CL) nació como mecanismo para estudiar funciones detalladamente. En 1893 Gottlob Frege (1848-1925) observó que para estudiar funciones de cualquier número de argumentos basta con concentrarse en funciones de un solo argumento. Tomemos la función suma + que toma los argumentos A y B (enteros no negativos) y produce el valor A + B. Definimos la función ⊕, de un solo argumento, que aplicada a un valor A retorna una nueva función ⊕ (A). Esta nueva función aplicada a un valor B produce como resultado A + B. Nótese que 0 no es aplicada simultáneamente a A y B sino más bien en forma sucesiva, (⊕ (A)) (B) = A + B.

Esta es la forma en que las computadoras operan. Todos los programas son funciones de un solo argumento. Un programa que computa la función A + B busca primero el valor de A en la memoria, luego hace lo mismo con B y finalmente computa la suma. Si la ejecución del programa se detiene después de que se extrae A de la memoria, el programa resultante corresponde a la función intermedia ©(A).

Aparentemente Frege no profundizó mucho su idea. Esta fue redescubierta independientemente por Schónfinkel en 1924 y por Curry en 1930, además de la increíble conclusión de que todas las funciones que tienen que ver con la estructura general de funciones pueden ser construidas a partir de dos funciones básicas K y S. Schónfinkel utilizó una notación diferente para la aplicación de funciones; denotó la aplicación F(A) como (F A), donde los paréntesis asocian a la izquierda.

En 1932, Alonzo Church (1903-) propuso que toda función F tal que F(x) = M, (donde M está compuesta a partir de los operadores K, S y una variable x) fuera denotada como F = λx***.M.*** M es (intencionalmente) parte del nuevo nombre de la función de tal forma que sea posible computar por inspección el resultado de aplicar F a cualquier valor. Para denotar la aplicación de una función Church introdujo la siguiente regla:

$$(\lambda.x.M)\ N = M\ [\ x := N]$$

donde M [x:= N] corresponde al resultado de substituir cada instancia de la variable x en M por N. La belleza de este método es que permite evaluar la aplicación de una función como un proceso de reducción; ésta fue la percepción formal de "computo" presentada por Church.

En los años 50 John MacCarthy trabajó en varias propuestas para implementar el Cálculo Lambda en el lenguaje de programación LISP. MacCarthy reconoció a los procedimientos como funciones de un solo argumento. En CL las funciones pueden ser aplicadas unas a otras así como devolver funciones como resultado.

En LISP es posible aplicar un procedimiento a otro y ocasionalmente se obtienen procedimientos como resultados de aplicaciones. Sin embargo el lenguaje de programación LISP no constituye una implementación total del CL; de hecho en ciertos aspectos es totalmente distinto al Cálculo Lambda. El lenguaje de programación SCHEME es la impementación más fiel al CL de Church.

En primera instancia el CL aparenta ser un sistema débil; sin embargo, como lo indicó Church en 1932, los números enteros no negativos pueden ser fácilmente definidos en CL. Kleene descubrió cómo efectuar definiciones recursivas en CL y poco a poco se definieron centenares de funciones enteras. En varias investigaciones no publicadas de Barkley Rosser se definieron tantas funciones que Church conjeturó que toda función algorítmica es definible en CL. Church y Rosser demostraron en 1936 que toda función definible en CL es algorítmica por lo que se enunció lo que conocemos como la "Tesis de Church".

La Tesis de Church

Para 1936 existían tres poderosos paradigmas que clamaban formalizar el concepto de función computable. Dado que muchas funciones enteras eran definibles en CL se enunció la siguiente hipótesis:

Las funciones calculables (algorítmicas) entre enteros no negativos y sí mismos corresponden exactamente a las funciones definibles en CL.

Como la noción de "calculable" es intuitiva, la hipótesis no puede ser sujeta a demostración. Lo que afirma la tesis de Chruch es que cierto concepto intuitivo corresponde a un objeto matemático específico. Es teóricamente posible que en un futuro se deje de utilizar la tesis de Church si es que alguien puede proponer un modelo alternativo de cómputo que cuente con los requerimientos de "labor finita en cada paso" y que además sea capaz de efectuar cómputos que no son posibles en CL o MT. Nadie considera esta posibilidad factible.

La tesis de Church recibió mucha aceptación cuando Kleene demostró en 1936 que su criterio de recursividad general era equivalente a CL. También fue en 1936 cuando Turing presentó su concepto de máquina abstracta (MT) como modelo matemático del concepto de cómputo. Pero en 1937 demostró que la noción de "computable por MT" es equivalente a ser definible en CL. Esto explica por qué el CL juega un rol tan importante en la teoría de programación de computadoras y diseño de lenguajes.

También en 1936, y de forma independiente, Post desarrolló ideas similares a las de Turing. Turing pu-

blicó primero por unos cuantos meses; más adelante en 1943, Post propuso otra definición de “efectivamente calculable” que resultó ser equivalente a las ya existentes. Aun así, en 1951, Markov propuso una nueva definición que también se demostró equivalente al CL.

Conclusión

En los párrafos anteriores hemos estudiado el desarrollo de la teoría de la computabilidad, también conocida como teoría de la recursividad. Como hemos visto, el concepto de recursividad tiene muchas virtudes (claridad y compacidad), pero también tiene "defectos".

Una de las cualidades de las computadoras electrónicas modernas, que no existe en las Máquinas Turing como han sido definidas, es la capacidad de ser programadas. Esto significa que necesitamos una MT para resolver un problema específico, mientras que las computadoras reales son de "propósito múltiple" y no ad-hoc para una aplicación dada. Sin embargo, en esencia, las MT son programables. Utilizando métodos de *codificación* es posible escribir descripciones de máquinas Turing como secuencias de símbolos (que pueden ser inscritos en la cinta de otra MT). Además, es posible diseñar un tipo especial de MT llamada la ***Máquina de Turing Universal (MTU)*** que recibe la codificación de una MT arbitraria (incluyendo la codificación de sí misma) y simula la máquina codificada aplicada a un valor de entrada también codificado. Esta interesante propiedad de las MT es fuente de un resultado todavía más sorprendente que la existencia de la MTU; se le conoce como *El Problema de Finalización.* Este resultado dice (en esencia) que no es posible construir una Maquina de Turing que decida si una MT arbitraria (con cierto valor de entrada) va a detenerse eventualmente (finalizar el proceso de cómputo). La prueba es bastante simple: supongamos que existe una máquina M1 que decide si otra máquina M va a detenerse al procesar cierto valor de entrada. M1 toma la codificación de M (y el valor de entrada), computa por cierto tiempo y eventualmente se detiene dejando en su cinta una indicación de "Sí" si M termina de procesar su entrada o "No" si M nunca termina de computar (entra en un ciclo infinito). Ahora podemos construir una máquina M2 que toma un valor de entrada y simula a M1 sobre éste; si M1 se detiene con un valor "Sí" entonces M2 entra en un ciclo infinito, si M1 se detiene con "No" M2 se detiene y termina de computar con cualquier información en su cinta. ¿Qué sucede si alimentamos a la máquina M2 con su propia codificación?:

- Si M2 se detiene es porque M1 decidió que M2 entró en un ciclo infinito.
- Si M2 entra en un ciclo infinito es porque M1 decidió que M2 se detenía eventualmente.

en ambos casos M1 está reportando la respuesta equivocada. Si tomamos el punto de vista de Church -que las MT corresponden a algoritmos, entonces el Problema de Finalización puede describirse de la siguiente forma:

> No existe un algoritmo para determinar si otro algoritmo va a detenerse o a entrar en un ciclo infinito. En otras palabras, todo algoritmo diseñado para resolver el Problema de Finalización es, intrínsecamente, incorrecto; es más, el contraejemplo puede ser construido a partir del algoritmo mismo.

Problemas como estos son denominados como *indecidibles* o *sin solución.* Existen muchos otros problemas no decidibles en la Matemática y las Ciencias de la computación. Muchos de ellos se derivan de la Lógica Matemática a través del trabajo de Gódel, Church y Turing. Los resultados de Gódel y Church se concentran en la no existencia de algoritmos para determinar la correctitud de teoremas en sistemas lógicos. Otros resultados provienen de la teoría de números y álgebra. Es realmente sorprendente cómo en las Matemáticas y las Ciencias de la computación es posible demostrar la existencia de problemas que no tienen solución.

En este momento la Teoría de la Computabilidad es de aplicabilidad limitada. Generalmente se preocupa por la existencia de métodos computacionales en vez de atacar preguntas sobre eficiencia y/o buen diseño. No obstante, el modelo general de cómputo elaborado por Turing y Church ha sido (y es) utilizado en múltiples ramas (compiladores, sistemas operativos, bases de datos, computabilidad, diseño de circuitos, etc..) de las ciencias de la computación teórica y práctica.

Bibliografía

Hopcroft J., Ullman J. *Introduction to Autómata Theory, Languages, andComputation.* Addison-Westluey, Mass. (1979).

Lewis H, Papadimitriou C. *Elements of the Theory of Computation.* Prentice Hall, New Jersey (1981).

MacNaughton, R. *Elementary Computability, Formal Languages and Autómata.* Prentice Hall, New Jersey (1982).

Rogers, H. *Theory of Recursive Functions and Effective Computability.* MIT Press, Mass. (1987).

Rosser, B. "Highlights in the History of Lambda Calculus", *Communications of ACM,* (1982) pp. 216-225.

Selman, A. *Complexity Theory Restrospectíve.* Springer-Verlag, New York (1990).