La Bella y La Bestia (perdón, Lógica e Informática)

María Manzano
Universidad de Salamanca

24 de octubre de 2002

Índice

Resumen

El propósito de este artículo es poner de relieve los profundos lazos existentes entre Lógica e Informática.

En la Introducción destaco los distintos niveles en los que podemos situarnos para describir el funcionamiento de un programa, haciendo especial hincapié en el hecho de que en cada uno de ellos utilizamos un lenguaje formal. Desde el nivel inferior, pisando tierra, en dodnde se sitúa el computador con su unidad central, su memoria y registos, hasta el nivel metateórico en donde los programas son los objetos de estudio, la lógica formal constituye la fuente inagotable de recursos teóricos.

En la segunda parte del artículo muestro cómo desde los inicios de la Informática la Lógica ha sido fuente de fecunda inspiración y de influencia técnica: Boole, Frege Gödel, Tarski, Chruch, Kleene, Herbrand, Robinson, Turing, Post, Chomsky, etc. no son sólo lógicos notables sino también responsables, directa o indirectamente, de las maquinas de registros (Turing),de la caracterización del poder de los algoritmos y de la computación digital y de la demostración de su equivalencia (Church, Kleene, Turing), de la distinción entre síntaxis y semántica en los lenguajes

de programación (Frege,Tarski), y de ciertos lenguajes de programacion como el LISP y el PROLOG. También es de destacar la influencia que los sistemas formales, llamados gramáticas (Chomsky), diseñados para analizar la síntaxis de los lenguajes naturales, tuvieron en el diseño de los compiladores.

Como contrapartida la Informática ofrece una tierra nueva, rica en problemas filosóficos que pueden ser investigados con la poderosa maquinaria de la lógica formal. Es en la Lógica de Programas en donde se lleva a cabo esta empresa fascinante, en donde el maridaje entre Lógica e Informática es mejor entendido y cultivado.

1. Introducción

No llores, mujer hermosa,
ningún mal has de temer.
Aquí estás para mandarme,
y yo, para obedecer.

1.1. Niveles posibles en la descripción del funcionamiento de un programa

Cuando un programa está funcionando en un ordenador, la descripción de lo que sucede puede hacerse desde varios niveles muy distintos entre sí, y todos ellos poseen y utilizan su propio lenguaje formal.

LENGUAJE MÁQUINA

En un nivel inferior, pisando tierra, está la memoria, la unidad central (*CPU*) y ciertos dispositivos de entrada y salida (*input-output*). La memoria está constituida por un número elevado de piezas, en el sentido físico y tangible del término, llamadas registros, que a sus vez almacenan bits, que son los átomos de la Informática. Físicamente un bit es un interruptor magnético, que puede estar en dos posiciones: abierto o cerrado. lo normal es llamar a estas posiciones 0 y 1. Estos no quiere decir que un ordenador almacene sencillamente números en este nivel inferior, es un forma de hablar, tampoco creemos que los átomos de la física sean ecuaciones. Podemos considerar que los registros de memoria son instrucciones cuya primera parte contiene el nombre de la instrucción y la segunda, la instrucción propiamente dicha: un puntero (*pointer*) que señala a otro registro de memoria. Dichas registros, por estar situados de una manera rígida en la memoria, poseen dirección (*address*), otro número, naturalmente. Cuando un programa está funcionando, la *CPU* ejecuta instrucciones consistentes en pasar de un registro a otro. En el denominado lenguaje máquina las instrucciones son simplemente números en binario, la primera parte es el nombre de la instrucción y la segunda la dirección del registro a donde debe saltar. Aunque antiguamente se programaba en estos lenguajes numéricos binarios, que describen bastante literalmente lo que de hecho está sucediendo en la máquina , nosotros al utilizar un programa no pensamos , ni de lejos, en estos términos. Hay una jerarquía de niveles y nosotros nos instalamos, todo lo cómodamente posible, en la cúspide.

LENGUAJE ENSAMBLADOR

Para describir lo que sucede al ejecutar un programa, también podemos situarnos en el nivel inmediatamente superior a usar el denominado lenguaje ensamblador (*assembly language*). No es un gran salto,pero en vez de dar los nombres de las instrucciones mediante números en sistema binario, y las direcciones de los registros a donde debe pasar mediante un código binarios, utilizamos

nombres en un sentido próximo a un lenguaje natural. El que se pueda describir lo que sucede en este lenguaje no tiene nada de particular, pero lo interesante es que también se pueden dar órdenes al computador en este lenguaje.

¿Qué sucede cuando se escribe un programa en lenguaje ensamblador? ¿Cómo puede el ordenador ejecutarlo?

Alguien tiene que haber escrito, en lenguaje máquina, un programa de traducción, un ensamblador (*assembler*), que la propia máquina pueda ejecutar para así entender y poder llevar a término la instrucción dada en ese lenguaje que no es el suyo. Esta es la idea fundamental que se explotará convenientemente en todo el proceso de abstracción hacia la cúspide , siempre ocasional, de esta jerarquía de niveles.

LENGUAJES DE ALTO NIVEL

Después de programar en este lenguaje durante un cierto tiempo, en los años cincuenta se hizo evidente que al formular algoritmos −descripciones precisas de procesos que han de ser entendidos y ejecutados sin ambigüedad− había ciertos patrones fijos que se repetían de continuo (*rutinas*). Es decir, los algoritmos parecían tener ciertos componentes de un nivel superior que permitiría describirlos de manera relativamente concisa y estética. Además de estas regularidades −las rutinas− hay otras más complejas denominadas subrutinas (*subroutines y procedures*). Parecía entonces claro y evidente que el poder añadir a un lenguaje de programación la habilidad de definir entidades de un nivel superior, usando las previamente definidas, y el poder referirse y apelar a ellas mediante un nombre era ya un paso gigantesco −y de hecho, así es. El programador ahora no estaría sujeto al uso de un repertorio limitado de instrucciones, podría diseñar sus propios módulos. Los nuevos lenguajes construidos con estas ideas se denominaron lenguajes compiladores (*compiler languages*), el primero de los cuales fue el Algol (*ALGOrithmic Language*), inventado en los años cincuenta. Por supuesto, hacía falta un programa −el compilador (*compiler*)−, ejecutable por el ordenador, que tradujera las instrucciones escritas en este lenguaje al lenguaje máquina.

También por esa época se inventaron los denominados intérpretes (*interpreters*), que como los compiladores traducían lenguajes de alto nivel a lenguaje máquina. La traducción y ejecución era ya simultánea y línea por línea. De esta clase es que el que se utiliza con el lenguaje de programación que más gusta a los que se dedican a la Inteligencia Artificial, el LISP (que significa LISt Processing, no como también se describe esta abreviatura; *Lots of Irritating Silly Parenthesis*).

La diferencia entre un intérprete y un compilador es que el primero acepta como input programas acabados −en Algol, por ejemplo− y produce como output una secuencia de instrucciones en lenguaje máquina, el intérprete está, por el contrario, funcionando continuamente y el leer, traducir y ejecutar es simultáneo.

A su vez estos lenguajes de programación se utilizan para escribir programas específicos de manejo de bases de datos, cálculos diversos, procesadores de textos

o de imágenes , aprendizaje de técnicas determinadas, expertos en bolsa o en medicina, juegos, etc. Este producto final es el que utiliza el usuario normal en su trabajo u ocio.

Hace unos años se pensaba que la gran diferencia entre los ordenadores y las máquinas y herramientas producidas por la industria tradicional residía en que los ordenadores descargaban de trabajo intelectual al hombre y las otras máquinas, de trabajo físico. Hoy en día no está nada claro: si yo puedo hacer la compra en El Corte Ingles utilizando mi moden −tras consultar la oferta de ese día−, regar el jardín con sólo darle a la tecla, reservar libros de la Biblioteca desde mi butaca, enviar artículos a revistas , o comunicarme con mis amigos sin hacer tediosas colas en Correos, y saber mi saldo −posiblemente negativo a causa de la factura de la telefónica− sin salir de casa, ¿no me he descargado de un trabajo físico notable?.

LÓGICA DE PROGRAMAS

Hay, por supuesto, otros niveles de mayor abstracción en los que podemos situarnos para intentar entender y analizar los programas. Podemos, desde fuera, crear un lenguaje formal en el que expresar propiedades de los programas y estaremos entonces haciendo lo que se denomina Lógica de programas. Eventualmente, estas investigaciones pueden dar origen a programas de verificación de programas, ejecutables por ordenador. Y así el propio ordenador se dice así mismo: *‹ ‹muy bien Pepe, muy bien››.*

Sobre la lógica de programas hablaremos luego, pues es en este nivel metateórico en donde nos vamos a situar.

1.2. Informática

La informática trata de los computadores y de la computación. A la informática de concierne la invención y análisis de los algoritmos, el diseño de programas y de lenguajes de programación para expresar algoritmos y la construcción de sistemas de computación, máquinas incluidas , para ejecutar o implementar (*implement*) los lenguajes de programación. Sin embargo, su objetivo es el de abstraerse de las particularidades de los mecanismos físicos que comportan sus circuitos internos para venir a fijarse en los formalismos usados en sus operaciones y programación. Es decir, hay que poner especial énfasis en el estudio de software, que a su vez, es creciente más y más abstracto.

Uno de los investigadores en Informática más conocido y reconocido, Goguen, afirma que deberíamos investigar las leyes del proceso de abstracción y conocer su teoría matemática y su metodología puesto que incluso los programadores en su trabajo práctico −no sólo teóricos− tienen que tratar con ella a un nivel sin precedentes. Para ilustrar su comentario de que la abstracción se ha convertido en una herramienta indispensable, cita los ejemplos de lo que se denomina sintaxis abstracta (*abstract syntax*) y tipos abstractos de datos (*abstract data types*), imprescindibles en la programación actual. La teoría general de la abstracción que él propone, su Teoría de las Instituciones [1] −que es una

teoría semántica– así como su complementaria, las Lógicas Generales de Meseguer [5] –que propone un marco en donde definir lógicas con cálculo y todo– están ambas basadas en la Teoría de las Categorías, una de las más abstractas que manejan en Lógica.

Es este énfasis en la abstracción, y por ende en el software, lo que confiere a la Informática una cierta continuidad, situándose por encima de los cambios tecnológicos que pudieran afectar a la construcción de los ordenadores. La informática ha de ser lo suficientemente abstracta como para poder encajar sin trauma el paso de los transistores de sílice a las modernas tecnologías ópticas. Los formalismos para representar y manipular datos, en la medida en que sean independientes de las máquinas y sus tecnologías de fabricación, son fundamentales en la Informática y especialmente mimados por los teóricos del ramo.

1.3. Informática teórica

La informática Teórica se plantea como objetivo el establecer y analizar los conceptos generales, las herramientas matemáticas y las técnicas heurísticas[1] y metodológicas que se emplean o pudieran emplearse en la Informática práctica. Además de ello, mediante sus explicaciones teóricas y su reflexión metateórica, busca el fundamentar científicamente a la Informática.

En la actualidad sucede que la distancia entre la Informática Teórica y la práctica se reduce constante y notablemente y que el diálogo entre ellas es continuo en los países avanzados y que debiera intensificarse en nuestro país. De hecho, uno de los quehaceres más frecuentes en Informática teórica es el de diseñar buenos programas de verificación de programas.

2. Lógica e informática

A la lógica le concierne el análisis y codificación del razonamiento; le interesa el significado y la transformación de los enunciados de nuestro discurso sobre el mundo. En la lógica formal se define con precisión un lenguaje artificial, se atribuye significado a los enunciados del lenguaje mediante modelos matemáticos, y se define un cálculo; esto es, un sistema de axiomas y reglas que rigen la transformación de los enunciados del lenguaje formal. Con estas reglas se reproducen mecánicamente los razonamientos válidos; es decir, los razonamientos que jamás nos llevarán de hipótesis verdaderas a conclusiones falsas. Dependiendo del nivel de abstracción que vayamos a necesitar y de la realidad a tratar, hay varios tipos de lógica: lógica sentencial, lógica de primer orden, lógica de segundo orden, teoría de tipos, lógica infinitaria, lógica multivariada, lógica polivalente, lógica temporal, lógica modal, lógica de programas, etc. De entre otras ellas es la lógica de primer orden la de mayor aplicabilidad.

Por otra parte, ha sido en la lógica matemática en donde se ha desarrollado la teoría de la Computabilidad, definido el concepto matemático de recursividad

[1] Un programa heurístico es capaz de automodificarse de «aprender de sus errores».

y caracterizado matemáticamente el poder de los algoritmos de la computación digital.

2.1. Lógica: Orígenes próximos

Centrándonos en la lógica de primer orden, los orígenes hay que buscarlos en la antigüedad clásica griega. Para ellos la lógica consistía fundamentalmente en la esquematización de los razonamientos válidos, destacando los silogismos aristotélicos.

A mediados del siglo XIX, Boole (1815-1864) creó el primer cálculo lógico, para la lógica sentencial.

La lógica en sentido moderno nace a finales del siglo XIX y principios del XX.

TEORÍA DE LA PRUEBA

Frege (1848-1925), Peano (1858-1932), Russell (1872-1970), Hilbert (1862-1943), Herbrand (1908-1931) y Gentzen (1909-1945) desarrollaron la *Teoría de la Prueba* de la lógica de primer orden. Todos ellos pretendían sistematizar el razonamiento matemático y atacar con la poderosa artillería lógica la fundamentación de la matemática.

Frege es el padre de la lógica moderna, al que debemos gran parte de las distinciones y conceptos en ella usados. El primer cálculo para la lógica de primer orden fue el *Begriffschrift* de Frege. Russell y Whitehead con su *Principia Mathematica* intentaron reducir los conceptos matemáticos −de la aritmética y el álgebra− a conceptos lógicos. Peano axiomatizó la aritmética.

La teoría de la prueba en un sentido mucho más delimitado nació con el denominado *programa de Hilbert*. La idea de Hilbert era la de explotar al máximo la naturaleza finita de las pruebas para proporcionar una fundamentación de la matemática. Podría resumirse su concepción diciendo que preconizaban una axiomatización de las teorías matemáticas de la que pudiera probarse su:

1. *Consistencia.* Es decir, que nunca se podrá demostrar como teoremas de la teoría una sentencia y su negación.

2. *Completud.* Es decir, que cada sentencia −del lenguaje en el que se axiomatizó la teoría− sea ella misma o su negación un teorema de la teoría axiomática.

3. *Decidibilidad.* Es decir, que exista un procedimiento efectivo mediante el cual, en un número finito de pasos, se determine si una sentencia del lenguaje es o no un teorema de la teoría.

Los sistemas de Cálculo de Gentzen orientaron la teoría de la prueba a sus actuales derroteros, ligada inexorablemente a la perspectiva informática. El teorema de Herbrand de 1930 y, posteriormente, el de Robinson se consideran los pilares de la *demostración automática de teoremas*.

TEORÍA DE MODELOS

En el nacimiento de la lógica de primer orden participan decisivamente otro grupo de investigadores cuya orientación apuntaba a la, posteriormente bautizada, *Teoría de Modelos.* Löwenheim (1878-1957), Skolem (1887-1963), Gödel (1906-1978) y Tarski (1901-1983) son los pioneros de otra línea de investigación consistente en el estudio de las estructuras matemáticas considerando las leyes a las que obedecen. Löwenheim y Skolem demostraron teoremas generales acerca de la infinita variabilidad de la cardinalidad de los modelos de las teorías de primer orden, de la incapacidad de esa lógica para caracterizar estructuras infinitas, de su negada habilidad para distinguir entre cardinalidades infinitas. Gödel demostró la completud del cálculo de la lógica de primer orden. A Tarski le debemos los conceptos fundamentales de la semántica y de la Teoría de Modelos. A él le cabe además el mérito de haber concebido y dirigido un programa de investigación sistemática en esta disciplina.

En 1931 Gödel[4] demostró que si la aritmética elemental es consistente, no puede ser completa, y que en general el programa de Hilbert es irrealizable. Para demostrar este teorema, conocido como *teorema de incompletud,* Gödel introdujo el concepto de recursividad.

Comentario 1 *Estamos usando el término completud de dos formas: (1) completud de una lógica y (2) completud de una teoría. En el primer caso es una propiedad del cálculo; a saber, que sea capaz de generar como teoremas a todas las fórmulas válidas. En el segundo caso es una propiedad de una teoría; a saber, la de ser tan potente que toda sentencia del lenguaje (o su negación) se derive de la teoría.*

TEORÍA DE LA RECURSIÓN

¿Cuándo decimos que una función es recursiva?, ¿Qué significa ser recursiva?

Hay varias definiciones precisas, entre sí equivalentes, de este concepto. La noción intuitiva correspondiente a ser recursiva es ser *efectivamente computable.*

¿Cuándo decimos que una función es efectivamente computable?

Sencillamente, cuando hay un procedimiento efectivo −esto es, un algoritmo− que la computa. Un procedimiento efectivo debe cumplir los siguientes requisitos:

1. Instrucciones precisas, de longitud finita. Estas instrucciones no deben requerir astucia ninguna por parte de la persona o máquina que las siga y no deben incluir el azar −lanzar una moneda al aire, por ejemplo.

2. Si se le da un argumento $\overline{x}$ en el dominio de la función, debe producir $f(x)$ en un número finito de pasoso de cálculo.

3. Si se le da un x que no esté en el dominio de la función, debe seguir indefinidamente o parar, pero en ningún caso producir un valor $f(x)$.

Como ejemplos de funciones efectivamente computables podemos citar la suma y la multiplicación. Los procedimientos efectivos de cálculo son los que aprendimos en el *«cole»*. Por otra parte, toda función con dominio finito es efectivamente computable más interesantes que las de estos ejemplos.

Nosotros no imponemos restricciones de naturaleza práctica a los procedimientos efectivos. Cuando se trata de funciones sobre los naturales, los argumentos de las funciones han de ser números naturales, pero no se restringe su tamaño. Por otra parte, aunque el número de pasos ha de ser finito, tampoco ponemos un tope, Además, no se prefija la cantidad de papel −o espacio de memoria− que haya de precisarse para realizar el cálculo. Estas consideraciones son importantes cuando comparamos la computabilidad efectiva con la práctica: la clase de las funciones computables la obtenemos en una situación ideal en donde no importa el tiempo ni la memoria requerida. Las funciones recursivas son las efectivamente computables, pero reservamos el vocablo *«recursiva»* para el concepto matemático; es decir, el definido con precisión.

Hay varias maneras equivalentes de definir la recursividad. Una versión mediante máquinas imaginarias −máquinas de Turing− fue concebida por Turing en 1936. Su trabajo es anterior a los computadores e influyó poderosamente en la creación e éstos.

Por esas mismas fechas Church se preguntó si el concepto de recursividad definido por Turing se correspondía con la noción intuitiva de computabilidad efectiva. Su respuesta, conocida como *tesis de Church,* es que sí. Church y Kleene definieron también el concepto de $\lambda-$definibilidad, que se corresponde −según mostró Turing− con el de recursividad.

El concepto de recursividad para funciones se puede generalizar para conjuntos; los llamamos decidibles. Hay conjuntos que son recursivos sólo a medias y los llamamos efectivamente −o recursivamente− enumerables: sus elementos pueden ser listados.

¿Por qué es la teoría de las funciones recursivas una parte de la lógica? Claramente, si no se hubiera inventado la teoría de las funciones recursivas en lógica, se habría hecho, más tarde, en informática. Sin embargo, no fue una casualidad histórica. Hay aspectos fundamentales de la lógica que requieren el concepto de procedimiento efectivo. De hecho, donde por vez primera se define con precisión este concepto fue en el artículo de Gödel de la incompletud de la lógica superior.

En primer lugar, la sintaxis de los lenguajes formales ha de ser decidible; esto es, debe haber un algoritmo que nos sirva para computar si una fila de signos es o no una fórmula. Como hemos visto, en la lógica se construyen cálculos para probar teoremas. Los cálculos han de ser tales que cualquiera pueda comprobar la corrección de la prueba mediante un procedimiento recursivo. El conjunto de las pruebas ha de ser recursivo y el de los teoremas del cálculo recursivamente enumerable. Por consiguiente, si sabemos que un conjunto de fórmulas no es recursivamente enumerable, sabremos que no hay cálculo capaz de generarlas y no tendremos que hacer esfuerzos inútiles para encontrarlo.

Por otra parte, una teoría axiomatizable −recursivamente, claro− tiene un conjunto recursivamente enumerable de teoremas. Si además de ser recursiva-

mente axiomatizable, es completa, entonces sus teoremas formarán un conjunto decidible −recursivo.

Estos son sólo unos cuantos ejemplos para subrayar que la teoría de la recursión es de la lógica una parte fundamental.

2.2. Usos de la Lógica en Informática

La lógica formal en el principal soporte matemático utilizado tanto en el diseño del hardware, como en los desarrollos de los lenguajes de programación y en la construcción de programas de Inteligencia Artificial.

¿Por qué juega la lógica este papel preeminente en la Informática?

Hay una forma radical de contestar a esta pregunta diciendo que la informática no es más que Teoría de la computación y que, por consiguiente, *es* lógica.

Por no quedarnos aquí, pensemos en el diseño del hardware y del software.

¿Por qué juega la lógica este papel preeminente en su diseño?

HARDWARE

En primer lugar, porque tanto nuestros ordenadores digitales −esto es, los que trabajan con circuitos codificados en sistema binario− como los que lo hacen con circuitos integrados *VLSI* (*Very Large Scale Integration*), todos utilizan lógica de algún tipo en su diseño. Lógica de Boole en el primer caso, lógica de orden superior, en el segundo.

LENGUAJES DE PROGRAMACIÓN

Además, los lenguajes de programación son lenguajes formales cuyas fórmulas bien formadas son los programas.

Por consiguiente, los resultados de índole general obtenidos en la investigación de los lenguajes formales son trasladables a los lenguajes de programación. Con frecuencia el propio lenguaje de programación está inspirado en algún cálculo o principio lógico, tal es el caso antes mencionado del *LISP*, inspirado en el cálculo Lambda de Church, o el del *PROLOG*, inspirado en el teorema de Herbrand y en el principio de resolución de Robinson.

SISTEMAS EXPERTOS

Por otra parte, un cálculo lógico puede, en determinadas circunstancias, ser programado, implementado en una máquina, con lo que sirve de base a un probador de teoremas. De esta manera también usamos la lógica para realizar cálculos mediante razonamiento automático al utilizar un sistema experto.

LOGICA DE PROGRAMAS

Finalmente, por ahora, para razonar acerca de los programas podemos utilizar un lenguaje formal. En la Lógica de Programas creamos un lenguaje en

el que se puedan expresar ciertas propiedades de los programas tales como la corrección de programas, la equivalencia de programas o la propiedad de tener fin −no inducir una computación interminable−. Es conveniente también el contar con un cálculo deductivo para verificar y controlar nuestros razonamientos acerca de los programas. En los casos en que el cálculo sea decidible, sucederá además que será programable y el razonamiento devendrá en razonamiento automático. Así la aspiración de Leibniz se verá colmada y en vez de razonar, calcularemos.

2.3. Influencia de la lógica en la informática

Desde los inicios de la Informática la Lógica ha sido fuente de fecunda inspiración y de influencia técnica: Boole, Frege, Gödel , Tarski, Church, Kleene, Herbrand, Robinson, Turing, Post, Chomsky, etc. no son sólo lógicos notables sino también responsables, directa o indirectamente, de las máquinas de registros (Turing) de la caracterización del poder de los algoritmos y de la computación digital y la demostración de su equivalencia (Church, Kleene, Turing), de la distinción entre sintaxis y semántica en los lenguajes de programación (Frege, Tarski), y de ciertos lenguajes de programación como el *LISP* y el *PROLOG*. También es de destacar la influencia que los sistemas formales, llamados gramáticas (Chomsky), diseñados para analizar la sintaxis de los lenguajes naturales, tuvieron en el diseño de los compiladores.

En la actualidad la influencia es también poderosa y han surgido las Lógicas de Programas y los lenguajes de programación lógica, ejemplos emblemáticos del diálogo entre Lógica e Informática.

2.4. Lógica de Hoare-Floyd: Las primeras lógicas de programas

En el año 1969 Hoare introdujo un método axiomático para probar la corrección de programas basado en un método anterior, el de Floyd. Su método ha tenido un gran impacto en el diseño y verificación de programas. También ha sido usado como un medio de especificar la semántica de los lenguajes de programación.

SEMÁNTICA DE PROGRAMAS

En lógica clásica −y muy especialmente en teoría de modelos− se parte de una realidad matemática conocida y para hablar de ella se introduce un lenguaje formal. El puente entre el lenguaje formal y la realidad matemática es la noción semántica de verdad y, basado en ella, la noción de consecuencia. El cálculo deductivo, un conjunto de reglas de deducción, es una réplica mecanizable del concepto intuitivo de consecuencia.

En el caso de la semántica de los lenguajes de programación, pensad que a diferencia de lo que sucede en la lógica clásica, especialmente si se la mira desde la perspectiva de la teoría de modelos, el formalismo y sus reglas constituyen la

realidad inmediata y en lo que hay que aplicarse es en buscarle una semántica, unos modelos. Es evidente que el modelo real de lo que sucede, el *«mundo»* al que se refieren las fórmulas de nuestros programas no es otro que el constituido por la *CPU*, su memoria y registros, pero este *«mundo»* es de una complejidad tal que rebasa los límites de un análisis matemático riguroso. A causa de esta complejidad, se han hecho ciertos intentos de recurrir a la teoría de sistemas , considerándose el holismo como la postura filosófica a adoptar[2].

Las lógicas de programas, al menos en sus origen, pretendían resolver el problema de la semántica de programas y, por consiguiente, el de las expresiones que expresan propiedades de programas. Para los filósofos esta situación −la de encontrarnos con unos sistemas formales de reglas y axiomas de un lenguaje de nuestra invención, pero sin semántica precisa− no nos resulta del todo ajena ya que plantea similitudes con la de la Lógica Modal en la que la definición de diversos sistemas modales precedió a la definición precisa de su semántica. No deja de ser curioso que, aunque fuera por otros motivos, la Lógica de Programas recurriera a la modal para solventar sus problemas.

El significado o la semántica de un lenguaje de programación es un asunto mucho más complicado que su sintaxis y rápidamente planteó preguntas acerca de la fundamentación del software.

SINTAXIS, SEMÁNTICA Y CALCULO

Una de las cuestiones primordiales era la de la correción parcial de un programa S respecto del input p y el output q, expresada mediante la fórmula $\{p\}\ S\ \{q\}$. Intuitivamente esta fórmula expresa que si p es verdadera antes de ejecutar el programa S y si el programa S termina, entonces q es verdad tras la ejecución de S. Ejemplos sencillos de fórmulas de corrección de programas son:

$$\{x = 0\}\ x : x + 1\ \{x = 1\} \tag{1}$$

$$\{x \geqq 0\}\ WHILE\ x \succ o\ DO\ x : x - 1\ OD\ \{x = 0\} \tag{2}$$

Tres preguntas fundamentales se plantean:

1. *¿Cuál es la sintaxis de los lenguajes en donde escribimos* S, p *y* q*?*
2. *¿Cuál es el significado de la fórmula* $\{p\}\ S\ \{q\}$*?, ¿cuándo decimos que esta fórmula es verdadera?*
3. *¿Podemos encontrar cálculos adecuados en los que demostrar* $\{p\}\ S\ \{q\}$*?*

La primera pregunta suele contestarse diciendo que p y q son fórmulas de primer orden escritas en el lenguaje de la aritmética. Los programas S están escritos en un lenguaje que suele incluir asignaciones −por ejemplo; $x := x+1$−,

[2] Ver, por ejemplo [6]

composición secuencial ($S_1; S_2$), condicionales ($IF\ b\ THEN\ S_1\ ELSE\ S_2\ FI$), programas del tipo while ($WHILE\ b\ DO\ S\ OD$) y otro tipo de construcciones tales como procedimientos recursivos, programas no deterministas, etc.

Para responder a la segunda hay diversos métodos. Hemos dicho que podemos considerar que el computador real −su *CPU*, memoria etc.− es el universo en donde interpretar la fórmulas $\{p\}\ S\ \{q\}$. Puesto que esto es imposible a causa de la complejidad del sistema, se plantean varias soluciones. La primera es reemplazar el computador real por la máquina abstracta; es decir, un objeto matemático con estados, estructura de control, etc. La semántica así obtenida se denomina *operativa* y su característica es que aparecen secuencias de computación; esto es, secuencias de estados que describen en la máquina abstracta la ejecución del programa. Las ventajas de este método sobre el *denotativo*, que veremos después, es que se mantiene próxima a la semántica intuitiva. Su desventaja es que no es suficientemente abstracta y es con frecuencia tediosa.

En la semántica denotacional se evita el uso de máquinas abstractas empleándose un modelo matemático con conjuntos, funciones y operadores (funciones de funciones); esto es, un modelo extensional en donde no aparecen secuencias de computación. Habitualmente estos modelos son finitos o contienen el modelo estándar de los naturales, son una extensión suya. Es decir, no es relevante el significado de las fórmulas en modelos arbitrarios. Los programas se interpretan como funciones (o relaciones) entre estados iniciales y finales sin usar estados intermedios y las funciones de funciones sirven para interpretar los operadores programas.

La tercera de las cuestiones es cómo generar cálculos en los que derivar fórmulas de corrección parcial, $\{p\}\ S\ \{q\}$. Por ejemplo, considerad que el lenguaje de los programas incluye: asignaciones ($x =: s$, donde s es un término de la aritmética), composición secuencial ($S_1; S_2$) y programas tipo while ($WHILE\ b\ DO\ S\ OD$). Un cálculo deductivo para este lenguaje incluirá el axioma.

$$\left\{p\frac{t}{x}\right\}\ x =: t\ \{p\} \tag{3}$$

la regla de la composición secuencial de programas

$$\frac{\{p\}\ S_1\ \{q\} \qquad\qquad \{q\}\ S_2\ \{r\}}{\{p\}\ S_1; S_2\ \{r\}} \tag{4}$$

y la regla de programas tipo while

$$\frac{\{p \wedge b\}\ S\ \{p\}}{\{p\}\ WHILE\ b\ DO\ S\ OD\ \{p \wedge \neg b\}} \tag{5}$$

Enfoques de este tipo son los que fructificaron a raíz de los trabajos de Floyd y de Hoare a finales de los años sesenta. El que ellos iniciaron, hace ya 20 años, es sin duda el programa de investigación en Informática teórica más importante que ha existido. Este programa de investigación ha sido enormemente prolífico

y son incontables los métodos −lógicas, se suelen llamar− de corrección de programas que han aflorado pues, frecuentemente, se diseñaba una lógica específica para cada tipo de programa −asignaciones, composición, programas tipo while, etc.− Desde los métodos particulares y relativamente rupestres, hasta las lógicas dinámicas o las temporales, todos pueden considerarse enmarcados en el programa de Hoare-Floyd.

Su programa de investigación está basado en la lógica formal y tiene la pretensión de encarar los fundamentos de la programación −hay quienes lo comparan al programa de Hilbert, y no pensando en su irrealizabilidad−. Su propuesta suele presentarse diciendo que definen la semántica de los lenguajes de programación axiomáticamente; esto es, mediante unas lógicas de corrección de programas. En estas lógicas el comportamiento de los programas se formaliza mediante fórmulas adecuadas y se usan los axiomas y reglas del cálculo de la lógica creada para demostrar la corrección de los programas, como un teorema de cálculo introducido.

El cálculo introducido retorna al software como un programa verificador de programas y así ayudar al programador para comprobar si su programa original, el escrito primitivamente en lenguaje de programación, se comporta como se esperaba. De este programa de investigación han surgido ideas muy interesantes, útiles y bonitas sobre construcción de lenguajes de programación lógica, y sobre especificación y verificación de software. También han contribuido notablemente a fomentar el interés por los métodos de la lógica formal en el diseño del software.

METATEORÍA DE LAS LÓGICAS DE PROGRAMAS

A mediados de la década de los setenta se empezaron a estudiar las propiedades metalógicas de estas lógicas, se intentaron demostrar teoremas de corrección y completud para ellas −no son propiamente teoremas de completud, pues no se prueba para todas las fórmulas y no interesan todos los modelos−. Enseguida se advirtió que el denominado método de Floyd-Hoare, era incompleto en el sentido de que no todas las fórmulas que expresan corrección parcial de programas −que sean verdaderas en los modelos estándar− podían derivarse con las reglas del método (*lógica*) propuesto. Esto originó una frenética actividad investigadora para tratar de solucionar el problema. En unos casos se aumentaba la potencia del cálculo, permitiendo pruebas de longitud infinita, en otros se añadían modelos de computación no estándar, manteniéndose la exigencia de que las pruebas fuesen finitas.

2.5. Lógicas dinámicas

Heredera del método de Hoare-Floyd, pero con la pretensión de proporcionar no sólo un método de corrección de programas, sino una lógica en donde expresar y demostrar propiedades de programas, sino una lógica en donde expresar y demostrar propiedades de programas en un sentido mucho más amplio, nació a mediados de los setenta, la lógica dinámica.

Hay una lógica dinámica sentencial, una de primer orden y una de primer orden interpretada.

A la lógica dinámica se la puede presentar de estas tres maneras:

1. Como una de las lógicas de programas, planteando los objetivos de este área y diciendo por consiguiente para qué sirve
2. Como una extensión de la lógica modal; ubicándola, pues, históricamente
3. Presentando su lenguaje, su semántica y su cálculo deductivo

LA LÓGICA DINÁMICA ES UNA DE LAS LÓGICAS DE PROGRAMAS

Como he dicho anteriormente, las lógicas de programas se crean para expresar y probar propiedades de los programas de computación. Nos interesa poder expresar las propiedades de corrección, equivalencia de programas y la de tener fin, entre otras. Interesa también tener un cálculo deductivo para verificar nuestros razonamientos sobre programas.

La filosofía de la lógica dinámica se suele resumir diciendo que un programa es un objeto dinámico, capaz de hacer pasar al computador de un estado a otro. Un estado puede concebirse como el contenido de los registros de memoria usados por el programa.

La idea fundamental de la lógica de programas es que como resultado de la ejecución de un programa se altera el valor de verdad de las fórmulas. El caso intuitivamente más sencillo es cuando tenemos variables y asignaciones (*un programa*). Una fórmula puede ser verdadera para ciertos valores, pero si estos valores se modifican mediante asignaciones en el transcurso del programa, la fórmula posiblemente modificará su valor de verdad. Cuando manejamos programas es conveniente usar variables que puedan ir tomando diversos valores sobre ciertos universos en cualquier momento de la computación, a diferencia de lo que sucede en matemáticas, una misma variable puede tomar distintos valores en el transcurso del programa y, en particular , distinto valor inicial y final. Por consiguiente, el valor de verdad de las fórmulas sufrirá variaciones.

¿Por qué no se utiliza la lógica clásica cuyos lenguajes y cálculos conocemos tan bien? La respuesta habitual a esta pregunta es que precisamos de una lógica en donde se pueda expresar el cambio de valor de verdad de una fórmula como resultado de una acción (*la ejecución de un programa en particular*), en donde la verdad pueda ser relativizada. En la lógica clásica la verdad de una fórmula en un modelo dado es absoluta. No sucede así en la lógica modal, en donde la verdad se relativiza. No es, por consiguiente, de extrañar que la lógica propuesta por este fin, la denominada lógica dinámica, sea una generación de la lógica modal.

LA LÓGICA DINÁMICA ES UN EXTENSIÓN DE LA LÓGICA MODAL

La lógica modal es la lógica de la necesariedad y de la posibilidad. Comienza su historia con una etapa primitiva, no sistemática, que incluye los trabajos de:

Aristóteles, los megáricos, los estoicos y algunos medievales. Los lógicos modales necesitaron crear un formalismo capaz de captar situaciones dinámicas, de relatividad la verdad. Desde un principio se vio la conexión entre nociones modales y temporales, siendo ya debatida por los megáricos y los estoicos. Para Leibniz son *necesarias* las sentencias verdaderas en todo mundo posible y *posibles* los son las que valen en alguno.

La etapa sistemática se sitúa en este siglo y comienza, más o menos independientemente, con Lewis, Łukasiewicz y Carnap. Ellos desarrollaron diversos cálculos modales, pero la semántica, aunque ampliamente debatida, necesitaba ser sistematizada. La etapa siguiente es calificada por Bull y Segerberg [2] de *«revolución industrial»* y corresponde a la de Kripke, Prior, Kanger y Hintikka. En esta etapa se define con precisión la semántica. Aunque se atribuye a Kripke la paternidad de los mundos posibles −que es como se conoce a la semántica modal−, parece ser que no fue ni el primero ni, por supuesto, el único que definió a los modelos modales en la forma en la que nos han llegado.

$$\mathfrak{A} = \left\langle W, R, \left\langle P^{\mathfrak{A}} \right\rangle_{p \in \Phi_0} \right\rangle \tag{6}$$

es un modelo de la lógica modal.

Es decir, los modelos de la lógica modal son sistemas formados por un conjunto W de mundos posibles o de estados, una relación de accesibilidad entre mundos ($R \subseteq W \times W$) y una representación adecuada de los signos del lenguaje clásico ($P^A \subseteq W$ para cada letra sentencial $p \in \Phi_0$).

En el lenguaje de esta lógica, que contiene los signos $[]$ y $\langle\rangle$ para la necesariedad y la posibilidad, escribimos $[]\,\varphi$. Esta fórmula expresa que φ es necesariamente verdadera, que es verdadera siempre. $[]\,\varphi$ será verdadera en un mundo $w \in W$ siempre que φ sea verdadera en todo mundo accesible desde w −esto es, tal que $\langle w, t\rangle \in R-$. Por otra parte, $\langle\rangle\varphi$ es verdadera en un mundo w si φ lo es en al menos un mundo accesible desde w.

La relación de accesibilidad recibe en lógica modal multitud de tratamientos originándose diversas teorías modales: *KT*, *KB*, *KT5*, etc. Si pretendemos captar como noción de necesariedad una noción fuerte, leibniziana, la relación de accesibilidad ha de ser de equivalencia y la teoría modal correspondiente será *S5*. Si la noción de necesario no es tan fuerte, podremos tener relaciones de accesibilidad que sean simplemente reflexivas, o simétricas, o transitivas y estaremos en las teorías *KT*, *KB* o *K4*. Por otra parte, cuando la necesariedad se interpreta temporalmente, la relación de accesibilidd debe de ser un orden.

En la actualidad, la lógica modal se ha diversificado enormemente y a los objetivos tradicionales han venido a sumarse: 1) problemas filosóficos muy elaborados (Fine), 2) una ampliación de la teoría de los modelos con nociones modales (Mortimer), 3) la interpretación de los operadores modales como demostrabilidad (Boolos, Solvay), 4) nuestra lógica dinámica (Andréka, Harel, Kozen, Meyer, Németi, Pratt) y 5) los planteamientos de la denominada *"gramática de Montague"*.

PRESENTACIÓN DE LA LÓGICA DINÁMICA

¿Cómo se modifica en la lógica dinámica la semántica de los mundos posibles? Como veremos, el esquema abstracto de la lógica modal y, en particular, los modelos de Kripke, le sientan perfectamente a la lógica dinámica. W será ahora el conjunto de los estados relevantes para los programas en consideración. Con cada programa b asociamos una relación binaria de accesibilidad de forma que el par $< s, t >$ está en ella si hay un estado t al que se llega desde s mediante el programa b; es decir, una computación que transforma el estado s en estado t. Así, un programa es concebido como un conjunto de pares de estados: iniciales y finales. Puesto que vamos a admitir programas no deterministas −es decir, en donde el estado inicial no determina unívocamente el estado final− la relación de accesibilidad no necesita ser una función.

Como es de suponer, asociando a cada programa una relación de accesibilidad, se tienen que manejar simultáneamente muchas de ellas y los que haremos es colocar dentro de $[]$ y de $\langle\rangle$ el programa concreto al que nos referimos; es decir, escribiremos $[b]$ y $\langle b\rangle$. Además de esta extensión de la lógica modal consistente en considerar conjuntamente diversas relaciones de accesibilidad, en lógica dinámica está permitida formar programas compuestos a partir de programas simples; es decir, las relaciones se combinan mediante unión, composición y *«loop»* para formar nuevas relaciones.

El lenguaje de la lógica dinámica que voy a presentar es el sentencial.

Lenguaje

Con las fórmulas y programas atómicos −conjuntos Φ_0 y Π_0− se construyen inductivamente las fórmulas y programas de la dinámica usando para ello los conectores $(\neg, \vee, \wedge, \rightarrow, \leftrightarrow)$, los operadores modales ($[]$ y $\langle\rangle$) , los operadores de programas ($\cup$ $*$ y ;) y el operador de test (?).

En primer lugar, tenemos fórmulas atómicas (todas las $p \in \Phi_0$). Además de eso, a partir de fórmulas cualesquiera, los conectores nos permiten formar nuevas fórmulas (el procedimiento es el normal). Mientras que los programas se combinan mediante sus operadores para formar nuevos programas (la unión y la composición de programas son operadores binarios, mientras que el *loop* lo es monario). A partir de una fórmula se forma un programa que es el test de esa fórmula ($?\varphi$). Y con una fórmula un programa y un operador modal se forma una nueva fórmula (por ejemplo, $\langle a\rangle\varphi$). Son estas últimas las fórmulas características de la lógica dinámica, las que nos permiten expresar propiedades de programas.

Significado intuitivo

$(a;b)$ significa «*Haz a seguido de b*»

$(a \cup b)$ significa «*Haz a o b*»

$a*$ significa «*Haz a un número finito, pero no precisado de veces*»

$[a]\,\varphi$ significa «*a es particularmente correcto respecto del OUTPUT* φ»

$\langle a\rangle\,\varphi$ significa «*a es totalmente correcto respecto del OUTPUT* φ»

Aunque sea éste el significado intuitivo de estas expresiones, la definición, mucho más precisa, no es ésta, sino una definición inductiva.

Semántica

Sea A un modelo de Kripke,

$$\mathfrak{A} = \left\langle W, R, \left\langle P^{\mathfrak{A}}\right\rangle p \in \Phi_0, \left\langle Q^{\mathfrak{A}}\right\rangle_{Q\in\Pi_0}\right\rangle \tag{7}$$

Inductivamente se define:
$\mathfrak{A}(\varphi)$ conjunto de estados donde φ es verdad
$\mathfrak{A}(a)$ pares de estados iniciales y finales
En particular,

$$\mathfrak{A}(a*) = \left\{ \begin{array}{c} \langle s,t\rangle \in W\times W \mid \exists k \exists s_0,..,s_k (s_0 = s \wedge s_k = t \wedge \\ \forall i_1 \leq i \leq k\, \langle s_{i-1}, s_i\rangle \in \mathfrak{A}(a)) \end{array} \right\}$$

$$\mathfrak{A}([a]\,\varphi) = \{s \in W \mid \forall t\ (\langle s,t\rangle \in \mathfrak{A}(a) \Rightarrow t \in \mathfrak{A}(\varphi))\}$$

La semántica es, como dije, una ampliación de la modal. Los modelos contienen un universo de mundos o de estados, una familia de subconjuntos del universo de estados y una familia de relaciones sobre mundos. Aquí, en vez de tener una relación de accesibilidad, tenemos una para cada programa atómico ya que los programas son concebidos como objetos dinámicos que permiten que el computador pase de un estado a otro; es decir, cumplen la misma misión que las relaciones de accesibilidad entre mundos en la modal. Al interpretar las fórmulas y programas del lenguaje se hace de forma que toda fórmula represente el conjunto de los estados en donde es verdad y que cada programa se interprete como el conjunto de pares de estados iniciales y finales entre los que el programa nos lleva.

En particular, la interpretación del loop es la menor clausura reflexiva y transitiva de la relación sobre la que se aplica.

Propiedades de programas

Usando los programas básicos y los operadores de programas se pueden construir programas nuevos y expresar propiedades de programas.

Por ejemplo:

$$IF \;\; \varphi \;\; THEN \;\; a \;\; ELSE \;\; b \equiv_{Df} ((\varphi?; a) \cup (\neg\varphi?; b)) \tag{8}$$

$$WHILE \;\; \varphi \;\; DO \;\; a \equiv_{Df} ((\varphi?; a)*; \neg\varphi?) \tag{9}$$

$$\langle a\rangle\varphi \wedge [a]\,\varphi \tag{10}$$

(a es totalmente correcto, en sentido fuerte)

$$\varphi \rightarrow [a]\,\Psi \tag{11}$$

(a es parcialmente correcto respecto $INPUT \;\; \varphi$ y $OUTPUT \;\; \Psi$).

$$\langle a\rangle\varphi \leftrightarrow \langle b\rangle\varphi \tag{12}$$

(a y b son equivalentes por lo que respecta a la condición φ)

Cálculo

Para la lógica dinámica sentencial hay un cálculo completo en sentido débil. Entre los axiomas de este cálculo se cuentan los de la lógica sentencial no modal, los que regulan el funcionamiento de la composición de programas y alguno de los de la lógica modal. A diferencia de lo que sucede en lógica modal donde los axiomas describen la relación de accesibilidad, los axiomas dinámicos reflejan las características de los operadores de programas. De entre ellos cabe destacar estos dos −que describen el funcionamiento del *loop*−, el segundo de los cuales es conocido como axioma de inducción.

A5) $\langle a^*\rangle\varphi \leftrightarrow (\varphi \vee \langle a\rangle\langle a^*\rangle\varphi)$

A8) $\varphi \wedge [a^*]\,(\varphi \rightarrow [a]\,\varphi \rightarrow [a^*]\,\varphi)$

Metateoría.

El cálculo de la lógica dinámica sentencial es correcto y completo, pero en sentido débil, pues el teorema de compacidad falla. Es decir, es completo para validez ($\models \varphi \Longrightarrow \vdash \varphi$ es siempre cierto) pero no es completo para consecuencia ($\Gamma \models \varphi \Longrightarrow \Gamma \vdash \varphi$ no siempre es cierto).

Este cálculo es también decidible, aunque el tiempo empleado en calcular si una fórmula es o no un teorema, crece exponencialmente con la complejidad de la fórmula.

Más allá de la cúspide

Al estudiar las propiedades metalógicas de las lógicas de programas nos situamos a un nivel de abstracción inmediatamente superior a cuando las usamos para demostrar propiedades de programas.

En este nivel doblemente metateórico podemos acometer el estudio y la comparación de las diversas lógicas de programas, en particular , por lo que respecta a su capacidad expresiva, y muy especialmente , por su potencia para demostrar la corrección de programas.

Referencias

[1] Goguen, J y Burstall, R. [1984] *"Introducing Institutions"*. En [3], pp. 221-256

[2] Bull, R. Segerberg, K. [1984]. *"Basic Modal Logic"*. En

[3] Clarke, E. y Kozen, D. editores. [1984]. *Proceedings of the Logics of Programming Workshop*, volumen 164 de Lecture Notes in Computer Science, páginas 221–256. Springer-VerlagGoguen, J.A. y Burstall, R.M. Lecture Notes in Computer Science. vol. 164. Springer-Verlarg. Berlín. Alemania.

[4] Gabbay, D y Guenthner, F. [84]. *Handbook of Philosophical Logic Handbook of Philosophical Logic* 2^{nd} edition [2001]. Kluwer Academic Publishers. Dordrecht. Holanda.

[5] Meseguer, J. [1989]. *"General Logics"*. Logic Colloquium´87, pp 275-331. North Holland. Amsterdan.

[6] Sloman, A. [1978]. *The computer revolution in philosophy*. The Harvester Press. Sussex.