Convergencias y divergencias en la noción de computación

Convergences and divergences in the notion of computing

Javier Blanco,* Pío García ** y Renato Cherini ***

Rastrear la genealogía y elucidar el alcance de la idea de computación tiene hoy un rol más fundamental que la mera exégesis, ya que pese a ser central en diversas disciplinas -o quizá precisamente por eso-, es una noción que no está fijada de manera clara y unívoca, no tiene un status ontológico definido y está constantemente bajo sospecha, sujeta a ataques desde diversos frentes. En este artículo ponemos en consideración algunas de las ideas que en la década de 1930 confluyeron en la definición de lo efectivamente computable y que dieron lugar a partir de la década siguiente a la construcción de computadoras electrónicas. Consideramos también algunas divergencias conceptuales que se produjeron a partir de la aparición de las computadoras electrónicas y del computacionalismo en filosofía de la mente, las cuales repercutieron en la comprensión de estos artefactos y de la nueva ciencia de la computación. Ofrecemos, para concluir, una caracterización de los sistemas computacionales que permite poner en perspectiva estas discusiones actuales, volviendo a poner el énfasis en la noción de programa, presente en todas estas ideas, pero indebidamente olvidada en algunas de las críticas.

Palabras clave: procedimiento efectivo, pancomputacionalismo, intérpretes, máquina universal

Tracking the genealogy of the idea of computing and elucidating its scope and limits is a much needed task that exceeds the mere exegesis. Despite the fact -or perhaps because of it- that this notion has become central in many disciplines, it is not clearly determined, it has no definite ontological status and it is constantly under suspicion and attacked from different fronts. In this article, we consider the confluence of ideas that in the 30s of the 20th Century rose to the notion of effective calculability and, on the following decade, to the construction of actual computers. Furthermore, we consider certain conceptual divergences that surfaced with the emergence of electronic computers and computationalism in the philosophy of mind, having important repercussions on the understanding of these artifacts and on the new discipline of computer science. Finally, we aim at understanding computing systems in order to introduce a new perspective on current debates, stressing the notion of program which although undoubtedly present in many of the original ideas, it has been unfairly forgotten.

Key words: *effective procedure, pancomputationalism, interpreters, universal machine*

* Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba. Correo electrónico: javier.o.blanco@gmail.com.
** Facultad de Filosofía y Humanidades, Universidad Nacional de Córdoba.
*** Instituto Nacional de Tecnología Industrial.

Introducción

La historia de la computación se suele describir como una convergencia de ideas provenientes de diferentes tradiciones. La propuesta de procedimiento efectivo de Turing (Turing, 1937) es considerada como una concretización privilegiada de esta confluencia. Así, en un trabajo de historia conceptual (Gandy, 1988), Robin Gandy muestra cómo el concepto de lo efectivamente computable aparece en la década del 30 a partir de la confluencia de ideas originadas en cuestiones lógico-matemáticas (paradigmático de esto es el *Entscheidung problem*), problemas en la fundamentación de la matemática (el programa de Hilbert y la necesidad de caracterizar los métodos finitistas) y de la reaparición de ideas desarrolladas un siglo antes por Charles Babbage en el desarrollo del "motor analítico", un mecanismo de cálculo programable cuya concepción prefiguraba las computadoras modernas (Davis, 2000).

Sin embargo, las preguntas acerca de qué es computar, cuándo un sistema computa, cuál es la relación entre computación abstracta y física, no desaparecieron a partir de la propuesta de Turing. Más aún, la filosofía de la mente y tanto la filosofía como las prácticas de la ciencia de la computación, son una fuente de problemas para las concepciones de la computación. En este sentido, así como se puede hablar de un momento de convergencia en la noción de lo efectivamente computable se puede hablar de un movimiento posterior de divergencia.

Esta situación puede explicarse de diferentes maneras. Una hipótesis plausible es que en la noción de computación efectiva no convergieron todas las ideas y problemas que guiaban a las tradiciones previas a la década del 30. Otra hipótesis posible es que muchos de los problemas generados luego de estos años fundantes responden a otras cuestiones que no tienen que ver directamente con la noción de computación. Finalmente, una tercera hipótesis es que la forma en que se suelen presentar algunos problemas sobre computación oscurece su vinculación con la noción de computación efectiva.

En este trabajo sugerimos que estas tres hipótesis explican parcialmente algunas de las formas en las cuales se ha problematizado la noción de computación. A su vez presentamos un análisis de la noción de intérprete presente tanto en la teoría de la computación como en el desarrollo práctico de software desde sus inicios, como una forma de generalizar la concepción de lo efectivamente computable y, en este sentido, aclarar algunas de las discusiones recientes en filosofía de la mente y de la ciencia de la computación.

1. La noción de computación efectiva: convergencia de ideas

Alrededor de la década de 1930 dos concepciones coherentes de lo efectivamente computable se anclaron en diferentes maneras de plantear el problema. Por un lado, se trató de caracterizar a los algoritmos como maneras mecánicas de resolver problemas. Por otro, en el marco de los sistemas axiomáticos, la pregunta estaba puesta en la noción de calculabilidad en una lógica. Para la gran mayoría de los participantes de esas discusiones (Church, Kleene, Herbrand y el mismo Gödel), la

noción de Turing de computabilidad resolvió ambas preguntas. La concepción de procedimiento efectivo mecanizado fue vista inmediatamente como un logro, ya que parecía una clara noción de qué significaba lo efectivamente computable (Sieg, 1994).

Una de las manifestaciones históricas de la caracterización algorítmica conformó la tradición mecanicista que incluye a los constructores de máquinas concretas de calcular. Lull, Pascal, Leibniz y Babbage realizaron sus ideas en dispositivos inferenciales y de cálculo. Para algunos de ellos -como Lull o Leibniz- el objetivo no era solamente lograr mecanizar el cálculo sino también representar de manera adecuada aquellos problemas sobre los cuales se pretendía una decisión certera y efectiva. Este aspecto de la tradición mecanicista es también precursor de los sistemas axiomáticos y aparece -aunque no se agota- en la codificación de la entrada de una máquina. En gran medida, la codificación de la entrada de una máquina universal de Turing puede verse como una generalización de este aspecto.

La otra tradición que confluye aquí es la que cobra cuerpo en el programa de Hilbert de fundamentación de la matemática. Frente a las antinomias que Russell encuentra en el intento fundacional de Frege, Hilbert plantea la posibilidad de una fundación axiomática de toda la matemática y la necesidad de pruebas absolutas de consistencia, inspirado por el éxito de la geometría axiomática, la cual él mismo se encargó de completar y demostrar consistente. Una pregunta asociada a este programa y central en el desarrollo de la computabilidad es el Entscheidung problem, el cual consiste en probar o refutar la existencia de un método mecánico para determinar si una sentencia de primer orden de la matemática es cierta o no.

El famoso teorema de incompletitud de Gödel (Feferman et al, 1986) dio un golpe de muerte al programa de Hilbert, o al menos acotó drásticamente sus posibilidades. La construcción de Gödel de una fórmula indecidible en un sistema con suficiente aritmética usa de manera esencial la posibilidad de interpretar un número como una fórmula, diluyendo en la práctica la frontera entre matemática y meta-matemática. En un trabajo reciente (Hehner, 1990), Erik Hehner muestra que, al menos implícitamente, la noción de intérprete es usada en dicho teorema, posibilitando incluso una versión más simple del teorema a partir de hacer explícita esta noción.

Si bien hubo varios intentos de caracterizar lo efectivamente computable y se demostraron equivalencias entre diferentes sistemas de candidatos, dando así forma a lo que luego se llamo la tesis de Church-Turing, fue recién cuando Turing pudo explicitar restricciones estrictas sobre qué significa un paso mecánico de cálculo (en principio realizado por un humano, como Turing mismo expresó y luego fue recordado por Wittgenstein) que se llegó a un consenso de que el problema estaba resuelto. Como consecuencia directa se resolvió, de manera negativa, el *Entshcheidung problem*. Una noción central al trabajo de Turing que tuvo importantes consecuencias conceptuales y prácticas es la noción de máquina universal. La máquina universal de Turing toma como parte de los datos de entrada una codificación de cualquier máquina de Turing (incluso, claro está, ella misma) y se comporta como esa máquina codificada con el resto de los datos. De esta manera se vuelven arbitrarias las fronteras entre el hardware, los programas y los datos. Puede verse entonces la

potencia del formalismo de las máquinas de Turing en este curioso hecho de poder definir un auto-intérprete.

2. Problemas con la noción de computación

Muchos problemas de las tradiciones que hemos reseñado muy brevemente en el apartado anterior encontraron su respuesta a partir de la propuesta de Turing. Pero otros aspectos y problemas de estas tradiciones permanecieron y se transformaron. En este trabajo proponemos que la noción de intérprete puede servir para entender y organizar muchos aspectos de la problematización de la noción de cómputo posterior a la década del 30. La centralidad de la noción de intérprete supone una toma de posición fuerte en discusiones actuales como la relación entre computación abstracta y física o la identificación y el ordenamiento jerárquico de sistemas computacionales. En este apartado presentamos las problematizaciones centrales de la noción de computación y relegamos el análisis de la noción de intérprete para la sección siguiente.

Distinguir entre la noción de efectivamente computable de Turing y las tradiciones previas permite explicar, al menos de manera parcial, la problematización posterior del concepto de computación. Entre estos problemas se puede destacar la relación entre computación física y abstracta y la identificación de un sistema computacional como tal. El primer problema incluye la cuestión de la naturaleza dual de los programas computacionales, la eventual bifurcación de la tesis Church-Turing para sistemas abstractos y concretos e incluso aspectos metodológicos como la relación entre verificación formal y testing de programas. En cuanto al segundo problema -la identificación de cuándo un sistema computa-, se puede señalar al pancomputacionalismo como la concepción que parece trivializar la noción de cómputo.

La tradición mecanicista muestra otro aspecto destacable. En ocasiones se cuenta la historia de la computación como si la construcción efectiva de una computadora física supusiera la elucidación previa de una noción abstracta de computación. Esta forma de defender la preeminencia de una concepción abstracta de computación se enfrenta con la evidencia histórica de la potencia expresiva y la flexibilidad que tenían los artefactos de Babbage. En particular, su máquina analítica podía ser programada. Esto sugiere que para construir una computadora no hace falta una noción abstracta previa de computación, pero sí parece ser necesaria dicha noción para reconocer a este artefacto como una computadora. Así, si bien la idea de Babbage de construcción de un artefacto computacional no tuvo asociada una teoría de lo computable, parece que esta fue necesaria para terminar de definir y reconocer a las computadoras como tales.

En relación con el segundo tipo de problemas que planteábamos más arriba, la tesis del llamado “pancomputacionalismo”, es una discusión que comenzó en el campo de la filosofía de la mente y luego fue tomada por la filosofía de la ciencia computacional. Un problema que aparece en cualquier definición de computación es saber si puede servir para distinguir sistemas que computan de aquellos que no.

Hinckfuss, Putnam y Searle (Putnam, 1987; Searle, 2004), entre otros, han elaborado argumentos para mostrar la trivialidad de las diferentes nociones de computación, la ausencia de rasgos específicos en la idea de computación que se expresan como pancomputacionalismo: "todo computa" a veces expresado como relativismo semántico: "la computación está en el ojo del observador".

Estas críticas toman diferentes formas. Quizá la mejor formulada es la que Putnam expresa en forma de teorema, en la cual muestra que cualquier autómata finito (sin entrada ni salida de datos) es implementado por cualquier sistema abierto con suficientes estados. La idea es que los estados abstractos pueden ser realizados por cualquier (clase de equivalencia de) estado físico. Si se tienen sistemas con entrada y salida, el teorema se vuelve más débil, pero igualmente sigue planteando cierta trivialización de la noción de cómputo: cualquier autómata finito será implementado por cualquier sistema abierto que exhiba la relación adecuada de entrada-salida.

3. Sistemas computacionales como intérpretes

Los sistemas pueden ser caracterizados en términos de sus posibles comportamientos. Por comportamiento entendemos una descripción de las ocurrencias de ciertos eventos considerados relevantes. Así, diferentes maneras de observar un sistema determinan diferentes conjuntos de comportamientos. Una definición más específica sólo tiene sentido dentro de algún marco de observación particular, lo cual no puede hacerse en el nivel de generalidad de esta propuesta.

Un caso particular de comportamiento utilizado frecuentemente para caracterizar cualquier tipo de sistema es la relación de entrada/salida. Una característica distintiva de los sistemas computacionales es, precisamente, la clase de comportamientos de entrada/salida que producen "sin cambiar un solo cable" (Dijkstra, 1988).

En otros trabajos (Blanco et al, 2010, 2011) hemos sugerido que la ubicua noción de intérprete (Jones, 1997; Abelson y Sussman, 1996; Jifeng y Hoare, 1998), aunque ligeramente generalizada, permite caracterizar los aspectos claves tanto de la ciencia de la computación teórica como aplicada.

Un intérprete produce un comportamiento a partir de alguna entrada, llamada programa, que lo codifica. Usualmente el programa depende de datos de entrada que, por simplicidad en este trabajo, los supondremos codificados junto con el mismo. En este sentido, la noción de intérprete es el vínculo necesario entre los programas que acepta como entrada (*"program-scripts"*) y los correspondientes comportamientos que produce (*"program-processes"*) (Eden, 2007).

Más precisamente, dado un conjunto B de posibles comportamientos y un conjunto *P* de elementos sintácticos, un intérprete es una función *i : P _ B* que asigna un comportamiento b a cada programa *p*. Se dice entonces que p es la codificación de *b*. Usualmente el dominio sintáctico de *P* se denomina "lenguaje de programación", y *p* se llama "programa".

Un sistema realiza un intérprete cuando cada vez que se le provee una codificación produce el correspondiente comportamiento observable. De manera más rigurosa, decimos que un sistema físico *I* realiza un intérprete *i* si es capaz de recibir un como entrada *p* y sistemáticamente producir un comportamiento b, tal que *i(p) = b*. Decimos entonces que *I* computa efectivamente *b* a través del programa *p*. En la noción de realización, los estados internos del sistema no son relevantes, y pueden darse de maneras diferentes.

Tanto en la teoría como en las prácticas de la ciencia de la computación el uso de intérpretes es ubicuo, aunque no siempre son presentados como tales. Por ejemplo:

- El "computador" presentado por Turing para describir sus máquinas. Es una persona equipada con lápiz y papel que toma una tabla de transición como codificación de un comportamiento (usualmente una función computable), cuyos datos de entrada están en una cinta, y aplica mecánicamente los pasos descriptos en esa tabla. Cada paso indica una posible modificación del contenido de la posición actual, un posible cambio de posición, y la siguiente instrucción. Si no hay instrucción siguiente, el programa termina.

- La máquina universal de Turing es descripta adecuadamente como un intérprete de cualquier máquina de Turing codificada en la cinta de entrada. Puede verse a la máquina universal como interpretando los comportamientos vistos como entrada/salida de cintas de caracteres o, componiendo con el intérprete anterior, directamente de funciones recursivas sobre números.

- El *hardware* de una computadora digital actual ejecutando su código de máquina es también un intérprete del conjunto de todas las funciones computables (tesis de Church). Tanto los programas como los datos se codifican en palabras de bits guardados en la memoria.

- Un *shell* de un sistema operativo (por ejemplo bash, en GNU/Linux) es un intérprete de las instrucciones de dicho sistema (como copiar archivos, listar un directorio, etc.) codificadas como secuencias de comandos primitivos.

- El ejemplo más común es un intérprete de un lenguaje de programación (como Perl, Haskell, Python, Lisp). Los programas y los datos están codificados por la sintaxis de dicho lenguaje. El conjunto de comportamientos está definido en la semántica de los lenguajes.

4. Jerarquías de sistemas computacionales

El concepto de intérprete permite responder de manera diferente a la pregunta sobre qué es computar y cuándo un sistema computa al posibilitar la especificación de niveles jerárquicos de computación. En su artículo (Eden y Turner, 2007), en un intento de dejar fuera algunos candidatos a lenguajes de programación anti-intuitivos, Turner y Eden determinan que un lenguaje de programación es tal sólo si es Turing-completo, es decir, si es capaz de codificar todas las funciones efectivamente

computables. Siguiendo esta línea, se podría pensar de manera simplista que un sistema es computacional si acepta un lenguaje de programación genuino. Sin embargo esta restricción parece demasiado fuerte, dado que algunos lenguajes de programación que no son Turing-completos son considerados genuinos por la comunidad, como por ejemplo: el lenguaje de demostradores de teoremas como Coq e Isabelle, que sólo permiten funciones que siempre terminan; algunos lenguajes derivados de teoría de tipos; o el lenguaje propuesto por el mismo Turner (Turner, 1995), que consiste en funciones primitivas recursivas de alto orden. Pero además, otros lenguajes considerablemente más simples como el lenguaje de una calculadora electrónica, parecen en principio lenguajes de programación interesantes, aunque menos potentes, y los sistemas que los soportan, sistemas computacionales en algún grado.

En lugar de dibujar una línea que separa sistemas computacionales de aquellos que no lo son, el concepto de intérprete permite distinguir distintos niveles de computacionalismo, colocando en el nivel más bajo los sistemas que realizan intérpretes con comportamientos triviales (una roca o un balde de agua), y en niveles superiores los sistemas que realizan intérpretes más complejos, como los que aceptan un lenguaje Turing-completo (una computadora digital).

Los diferentes niveles de computacionalismo se pueden establecer a partir de, al menos, dos jerarquías relacionadas a sendas nociones de la teoría de la computabilidad: simulación e interpretación. La jerarquía de simulación relaciona sistemas computacionales de acuerdo a su "poder computacional", esto es, el conjunto de computaciones que producen. La jerarquía de interpretación define una relación más fuerte que, bajo ciertas condiciones, implica la anterior y además caracteriza la noción conocida como "implementación".

La jerarquía de simulación se define en términos de un simple pre-orden entre intérpretes, considerando los conjuntos de comportamientos subyacentes a cada uno de ellos. Si *i* e *i'* son intérpretes definidos sobre los conjuntos de comportamientos *B* y *B'* respectivamente, decimos que *i* se comporta al menos como *i'* si *B'* es un subconjunto de *B*.

Normalmente los sistemas computacionales se comparan a través de los lenguajes de programación que aceptan, ya que suelen construirse explícitamente de acuerdo a ellos, en lugar de los comportamientos subyacentes que permanecen implícitos. La jerarquía de simulación puede redefinirse haciendo foco en los lenguajes de programación. Para ello es necesario introducir la noción de "función de compilación", que consiste en un mapeo que traduce un programa *p* en otro *p'* equivalente, posiblemente escrito en un lenguaje de programación diferente. Más precisamente, si *i* e *i'* son intérpretes sobre comportamientos *B* y *B'* y lenguajes de programación *P* y *P'* respectivamente, decimos que *c : P' _ P* es una función de compilación (entre *P'* y *P*) si para cada *p'* en *P'* se cumple que *i(c(p')) = i'(p')* y decimos que *i* simula a *i'*.

Ambas formulaciones son equivalentes, ya que es posible demostrar que el intérprete *i* se comporta al menos como *i'* si y sólo si existe una función de compilación entre *P'* y *P*. Sin embargo, la segunda formulación tiene su importancia ya que los

lenguajes de programación y la traducción entre ellos juegan un rol fundamental en las prácticas de la ciencia de la computación. En la literatura, el concepto de compilador se propone como central para el diseño y la comparación de lenguajes. En nuestro esquema, un compilador es un programa que codifica una función de compilación, es decir, un procedimiento efectivo que traduce un programa escrito en cierto lenguaje en otro. Se puede argumentar que la noción de compilador puede reemplazar al concepto de intérprete para identificar un formalismo sintáctico como lenguaje de programación; sin embargo, el concepto de intérprete es más primitivo ya que siempre es necesario un intérprete de un lenguaje para que los programas escritos en él puedan ser ejecutados. Mientras que un compilador posee cierta capacidad explicativa y permite comprender un lenguaje de programación en términos de otro lenguaje (más primitivo), es un intérprete el que finalmente caracteriza los comportamientos que el lenguaje es capaz de codificar.

La jerarquía de simulación establece cuando un intérprete puede comportarse como otro, pero no especifica cómo se lleva a cabo esa simulación. No es necesario que la función de compilación se pueda efectuar de manera mecánica, ni que sea una función efectivamente calculable. Requerir que el paso de traducción sea ejecutado por el mismo intérprete determina una jerarquía más fuerte que hace foco en la programabilidad de los sistemas involucrados y que denominamos jerarquía de interpretación.

Sean *i* e *i'* dos intérpretes, e *int* un programa particular de *P* que toma como entrada elementos (sintácticos) de *P'*. Si para todo programa *p'* en *P'* se satisface que *(i(int))(p') = i'(p')*, decimos que *int* es un programa-intérprete para *P'* y que *i* interpreta a *i'*. En el caso que *i* e *i'* sean el mismo, decimos que *i* es auto-interpretable, que es una conocida propiedad de los sistemas Turing-completos. Aunque el nombre elegido para referir al programa *int* puede generar confusiones en un comienzo, su elección no es arbitraria. La composición del programa *int* con el intérprete *i* puede ser considerada una realización del intérprete *i'* de acuerdo a los conceptos introducidos en la sección anterior.

Mientras que la jerarquía de simulación determina una relación de "ser parte de" entre los comportamientos, la jerarquía de interpretación caracteriza una relación de "ser elemento de", en el sentido que el conjunto completo de comportamientos de un intérprete es uno de los tantos posibles comportamientos del otro. Como consecuencia, si *i* interpreta a *i'*, no sólo puede *i* comportarse finalmente como *i'*, sino que su lenguaje de programación es capaz de codificar en un único programa el comportamiento completo del sistema *i'*.

Consideraciones finales

La noción de intérprete, como la hemos presentado, conlleva entre otras cosas una preeminencia de los aspectos que pueden llamarse teóricos de la noción de computación. Esta preeminencia se refleja en al menos tres aspectos:

• En primer lugar, la propuesta de máquinas universales constituyó una ampliación importante en la concepción de las máquinas. Descartes argumentaba que la diferencia entre las capacidades intelectuales del hombre y la de las máquinas radicaba justamente en la particularidad ineludible de todo artefacto físico que contrastaba con la universalidad del pensamiento humano.

• En segundo lugar, con la misma constitución de la concepción de computación se comenzaron a descubrir los límites de la misma. Dichos límites no dependían de los aspectos físicos de la computación, sino de restricciones teóricas.

• Finalmente, la noción de computación supone que el sistema computacional tiene la propiedad disposicional de ser programable.

La explicitación de las ideas, problemas y motivaciones de las tradiciones que convergieron en la década del 30 del siglo pasado permite en gran medida explicar las discusiones actuales sobre la noción de computación. Lo que hemos presentado de manera general como la relación entre computación abstracta y concreta y la tesis del pancomputacionalismo son dos ejemplos de este proceso. En este sentido, se puede decir que la convergencia no fue completa. Este resultado parece natural y esperable dada la complejidad de ideas que se desarrollaron durante un período de tiempo considerable en tradiciones muy disímiles. En este trabajo no hemos intentado dar cuenta de dicha complejidad, sino sólo señalar algunos hitos a partir de la noción de procedimiento efectivo de Turing. Este objetivo se complementó con nuestra propuesta de que la idea de intérprete puede servir, por un lado, para comprender la discusión a partir de la década del 30 sobre computación y, por otro lado, para aclarar y organizar dichas discusiones.

Las principales ventajas de nuestra propuesta (las cuales no fueron todas desarrolladas de manera comprensiva en este trabajo) son:

• Se establecen condiciones mínimas para que un sistema sea computacional en cierto grado. Estas condiciones no dependen de la tecnología actual.

• Se difumina la distinción entre software y hardware, uno de los mitos denunciados por Moor (Moor, 1978), lo cual es coherente con nuestra posición ontológica y con las prácticas de la ciencia de la computación (programar una máquina virtual o una "real" es transparente al programador; ciertas operaciones de bajo nivel a veces están implementadas en *hardware*, otras en microcódigo, otras directamente en *software*).

• Dado que la composición de intérpretes o aun de un compilador con un intérprete da como resultado otro intérprete, esta noción puede ser usada en diferentes niveles de abstracción; es más, funciona como el vínculo necesario entre estos diferentes niveles.

• Algunas cuestiones ontológicas (¿qué es un programa?, ¿cuándo un sistema es computacional?) pueden ser planteadas en términos más precisos admitiendo por ello respuestas más claras.

• Los intérpretes pueden ser usados para caracterizar sistemas computacionales, pero también para relacionar diferentes sistemas, estableciendo una jerarquía de sistemas computacionales. Usando esta jerarquía, la pregunta acerca de si un sistema es computacional o no puede ser reformulada preguntando qué conjunto de comportamientos implementa dicho sistema. A partir es esto es posible comparar el grado de computación de diferentes sistemas.

Bibliografía

ABELSON, H y SUSSMAN, G. J. (1996): *Structure and Interpretation of Computer Programs,* MIT Press, Cambridge, MA, 2ª edición.

BLANCO, J., CHERINI, R., DILLER, M. y GARCÍA, P. (2010): "Interpreters as computational mechanisms", 8th Conference on Computing and Philosophy (ECAP), octubre.

BLANCO, J., CHERINI, R., DILLER, M. y GARCÍA, P. (2011): "A behavioral characterization of computational systems", 1st Conference of the International Association on Computing and Philosophy (IACAP), julio.

DAVIS, M. (2000): *The Universal Computer: The Road from Leibniz to Turing,* W.W. Norton & Company.

DIJKSTRA, E. W. (1988): "On the cruelty of really teaching computing science", circulación privada, diciembre.

EDEN, A. H. (2007): "Three paradigms of computer science", *Minds Mach,* 17(2), pp. 135-167.

EDEN, A. H. y TURNER, R. (2007): "Problems in the ontology of computer programs", *Applied Ontology,* 2(1), pp. 13-36.

FEFERMAN, S., DAWSON JR, J. W., KLEENE, S. C., MOORE, G. H., SOLOVAY, R. M. y VAN HEIJENOORT, J. (eds.) (1986): *Kurt Godel: collected works, Vol. 1: Publications 1929-1936,* Nueva York, Oxford University Press, Inc.

GANDY, R. O. (1988): "The Confluence of Ideas in 1936", en R. Herken (ed.): *The Universal Turing Machine: A Half-Century Survey,* Oxford, Oxford University Press, pp. 55-111.

HE, J. y HOARE, C. A. R. (1998): "Unifying theories of programming", en E. Orlowska y A. Szalas (eds.): *RelMiCS,* pp. 97-99.

HEHNER, E. C. R. (1990): *Beautifying Godel,* Nueva York, Springer-Verlag New York, Inc., pp. 163-172.

JONES, N. D. (1997): *Computability and complexity: from a programming perspective,* Cambridge, MA, MIT Press.

MOOR, J. H. (1978): "Three myths of computer science", *British Journal for the Philosophy of Science,* 29(3), pp. 213-222.

PUTNAM, H. (1987): *Representation and Reality,* MIT Press.

SEARLE, J. (2004): *Is the brain a digital computer?*

SIEG, W. (1994): "Mechanical procedures and mathematical experience", en A. George (ed.): *Mathematics and Mind: Papers from the Conference on the Philosophy of Mathematics held at Amherst College,* 5-7 de abril de 1991, pp. 71-117.

TURING, A. M. (1937): "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proc. London Math,* s2-42(1), pp. 230-265, enero.

TURNER, D. (1995): "Elementary strong functional programming", en P. H. Hartel y M. J. Plasmeijer (eds): *Functional Programming Languages in Education, First International Syposium,* FPLE '95, Nijmegen, Holanda, 4-6 de diciembre, Proceedings, volume 1022 of Lecture Notes in Computer Science.