EPISTEMOLOGÍA E HISTORIA DE LA CIENCIA

SELECCIÓN DE TRABAJOS DE LAS XXII JORNADAS
VOLUMEN 18 (2012)

Luis Salvatico
Maximiliano Bozzoli
Luciana Pesenti

Editores

ÁREA LOGICO-EPISTEMOLÓGICA DE LA ESCUELA DE FILOSOFÍA
CENTRO DE INVESTIGACIONES DE LA FACULTAD DE FILOSOFÍA Y HUMANIDADES
UNIVERSIDAD NACIONAL DE CÓRDOBA


OCA
OFICINA DE
CONOCIMIENTO
ABIERTO | UNC

¿Cuándo un sistema (no) computa?

Javier Blanco[*] *y Renato Cherini*[◊]

Introducción

La pregunta que encabeza este trabajo puede reformularse de diversas maneras. Para la discusión que generó el pancomputacionalismo en filosofía de la mente, era lo mismo caracterizar la computación como tal que identificar cuándo un sistema computaba. Inclusive parecía que la tarea de identificar sistemas físicos que computan tenía una preeminencia sobre el problema teórico de caracterizar qué es computar. Pero el fracaso en la consecución de la primer tarea no implica que no se pueda llevar adelante la última. Es más, en este trabajo suponemos que una adecuada caracterización de la noción de computación es una condición para toda discusión posterior acerca de cómo identificar sistemas que computan, aunque ambas tareas son, como ya lo dijimos, diferentes.

En el este trabajo presentaremos un esbozo de caracterización de la noción de computación a partir de la de intérprete y luego sugeriremos una forma de vincular esta caracterización con el problema del pancomputacionalismo. En particular proponemos que la relación de implementación -entre sistemas computacionales abstractos y sistemas físicos- debería entenderse como una relación entre relaciones, en donde tienen una cierta preeminencia los aspectos prescriptivos que se definen explícitamente en el programa que genera los comportamientos del sistema computacional.

Computación efectiva

Para Turing (Turing, 1937) las reglas que determinan el proceso a computar son seguidas por un "computor" (él usaba la palabra "computer", pero la reservamos para su uso actual), es decir, una persona actuando mecánicamente. Computar es entonces seguir reglas, las cuales en el caso de las máquinas de Turing son tan elementales que ya no es posible ni necesario descomponerlas más.

La idea de efectividad permeó las discusiones en la década del '30. Turing justifica a partir de un análisis conceptual que lo que puede ser calculado por un ser humano abstracto de manera rutinaria es computable. La manera rutinaria de proceder requiere de una prescripción de los pasos a seguir. Para Turing, dadas las limitaciones perceptivas y de memoria, la manera rutinaria de proceder requiere reglas explícitas muy simples[i]

Gandy (Gandy, 1988) extiende esta idea a la computación por medio de mecanismos no-humanos, caracterizando abstractamente dichos mecanismos y determinando cual es el máximo poder de cómputo que puede obtenerse. Uno de los objetivos explícitos del trabajo de Gandy es independizarse de cualquier formalismo algorítmico particular, presentando sólo restricciones estructurales con las cuales es posible demostrar los límites de lo computable por esos mecanismos. De alguna manera, lo que se obtiene es un meta-formalismo que intenta capturar los diferentes formalismos para definir mecanismos computacionales, en particular las restricciones que deben satisfacer para ser considerados mecánicos. En este sentido, el trabajo extiende la noción de computación humana que

[*] U.N.C., blanco@famaf.unc.edu.ar
[◊] U.N.C., I.N.T.I., rcherini@inti.gob.ar

Turing plantea. La noción de programa no aparece en este trabajo que se concentra en describir los mecanismos sin fijar un formalismo particular (en el cual podría delimitarse qué se entiende por programa).

Si asociamos la noción de cómputo con la de programa, es decir, una computación sería un comportamiento producido por una prescripción (o programa, en un sentido general), dicho vínculo debe aparecer tanto en mecanismos "abstractos" (máquinas de Turing, Lambda cálculo, etc.) como en sistemas físicos concretos. Uno de los errores del pancomputacionalismo (Putnam, 1987; Searle, 2004) consistió en relacionar sólo los comportamientos abstractos con los concretos, no el vínculo entre comportamiento y programa. La noción de implementación sería entonces una relación de relaciones (o una función entre relaciones). La existencia determinante de dicho vínculo como explicativo de la idea de computación permite una caracterización homogénea de conceptos de la teoría de la computación y de las prácticas de la programación.

Programas e intérpretes, codificaciones y comportamientos

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 elaboraremos aquí.

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; Blanco et al, 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 en 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 $i: P \rightarrow B$ 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.

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 "computor" 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.

El concepto de intérprete sirve como criterio para distinguir entre sistemas que podrían ser computaciones (respecto a ciertas entradas y comportamientos) de aquellos que no. Dado que nos interesa capturar qué hace que un sistema sea programable, no asumimos ninguna tecnología de implementación en el concepto de intérprete. Los diferentes modelos computacionales, como las máquinas de Von Neumann, máquinas paralelas, computadoras de ADN, computadoras cuánticas, etc., pueden ser consideradas intérpretes porque producen sistemáticamente comportamientos a partir de sus codificaciones en un lenguaje predefinido.

Las nociones de intérprete y programa pueden ser vistas como relacionales, esto es, un intérprete es tal cuando es capaz de producir comportamientos a partir de programas; un programa es una estructura sintáctica capaz de ser interpretada. Un programa es tal sólo en relación a un intérprete dado y un intérprete es tal solo para un lenguaje de programación en particular. Los conceptos de programa, lenguaje de programación e intérprete son entonces relacionales e interdefinibles.

El hecho de que los programas sólo puedan definirse de manera relacional (cualquier cosa puede ser un programa) parece haberlos puesto en una posición subordinada a la hora de definir cuándo un sistema es computacional. Las caracterizaciones que aparecen en la literatura evitan referirse a los programas y tratan de caracterizar contextos mecánicos o funcionales que admitan la posibilidad de programación, posiblemente debido a que no disponen de una definición adecuada de programa es esos contextos. La noción relacional usada aquí parece resolver varios problemas.

La principal característica de un intérprete es que es programable: existe una sintaxis con la cual se puede codificar una variedad de comportamientos. El grado de programabilidad de un intérprete está dado por la variedad de comportamientos que el lenguaje de programación subyacente es capaz de codificar. El grado de programabilidad es

la característica distintiva de un sistema computacional interesante. Si consideramos que un sistema es computacional cuando es programable, entonces ser computacional es una propiedad que puede ser establecida sólo en relación a un conjunto de comportamientos y una codificación correspondiente. En otras palabras, la propiedad de ser computacional no tiene sentido independientemente de un conjunto de comportamientos y una codificación.

La propiedad de ser un intérprete para un conjunto dado de comportamientos puede ser satisfecha por ciertos sistemas. Un intérprete es una noción general que puede ser utilizada para caracterizar mecanismos físicos (computadoras, calculadoras), un humano actuando mecánicamente (el "computor" de Turing, un humano realizando las reducciones de un término lambda), formalismos matemáticos (la máquina universal de Turing), o computadoras que están más allá de la computabilidad de Turing (computadoras oráculos, etc). Mientras que es necesaria una contraparte (física) para la realización de un intérprete, la propiedad de ser un intérprete y concomitantemente, la propiedad de ser un sistema programable, puede ser determinada por su descripción abstracta

Implementaciones. Revisión de la noción de naturaleza dual de los programas

Encontramos en la literatura diversas expresiones de la dualidad de los programas, en tanto estos pueden verse como entes abstractos o como realizaciones físicas. La ubicuidad actual de las computadoras refuerza la relevancia de esta segunda mirada, mientras que la práctica de la ciencia computacional (tanto la teórica como la aplicada) parece enmarcarse en la primera, ya que los programadores trabajan creando entes abstractos, programas, los cuales pueden trasladarse de una máquina a otra, incluso desde el papel a la computadoras sin perder la identidad que el programador les adscribe. El concepto de implementación, usado profusamente en ciencia de la computación, puede dar cuenta de la relación entre los dos modos de existencia de los programas, pero, como bien remarcan Eden y Turner en (Eden and Turner, 2011) este mismo concepto está lejos de estar claramente definido.

Otra distinción que algunos autores toman como fundamental y que suele generar confusiones en su relación con la antes mencionada, es la que se establece entre *program-script* y *program-process*, es decir, el programa considerado como un texto o el proceso de computación al cual da lugar. Para autores como Eden (Eden, 2007) los procesos son objetos temporales, causales y si bien no son objetos físicos, son dependientes de la existencia de un sustrato físico que los ejecute. La relación entonces con los objetos abstractos que serían los programas necesita ser explicada. Eden pasa a asumir una solución monista en la cual los programas no serían abstractos sino que deberían ser pensados como objetos naturales, como una cadena de ADN, por ejemplo. Más allá de lo atractivo que pueda parecer dicha analogía, no parece condecirse con la manera en que los programas son considerados tanto en la teoría como en la práctica de la programación. Elaboraremos más esta idea.

La siguiente cita de Piccinini (Piccinini, 2010) pone de manifiesto algunas de las dificultades con el concepto de implementación:

> The problem of computational implementation may be formulated in a couple of different ways. Some people interpret the formalisms of computability theory as defining abstract objects. According to this interpretation, Turing machines, algorithms, and the like are abstract objects. But how can a concrete physical system

> implement an abstract object? Other people treat the formalisms of computability theory simply as abstract computational descriptions. But how can a concrete physical system satisfy an abstract computational description? Regardless of how the problem of computational implementation is formulated, solving it requires an account of concrete computation — an account of what it takes for a physical system to perform a given computation.

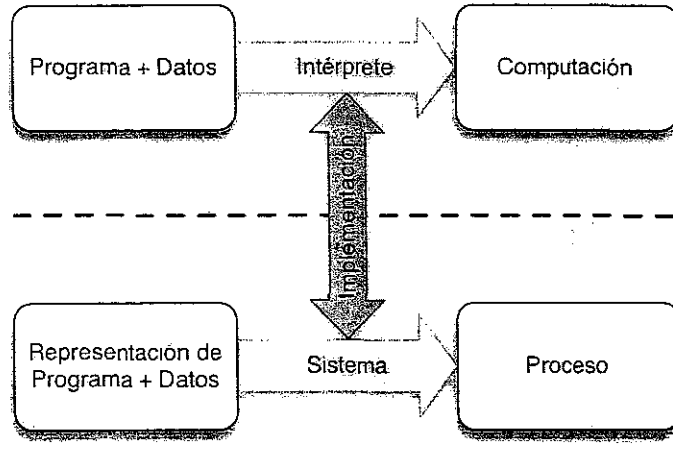
Luego de la cita, intenta dar cuenta de diferentes propuestas para dar contenido a esta noción, las cuales han sido ampliamente discutidas en la literatura. Parecería que aún se está lejos de llegar a un consenso acerca de una "solución" al problema de la implementación de sistemas computacionales, incluso es difícil evaluar las ventajas y desventajas de cada uno. Quizá el ejemplo más paradigmático y contra el cual vamos a contrastar nuestra propuesta es el simple *mapping account* que introduce y desarrolla sobre todo Putnam en (Putnam, 1987).

Según esta idea, un sistema físico S realiza la computación C cuando pueden ponerse en correspondencia los estados de S (de acuerdo a alguna descripción física) con los estados de la descripción computacional C, de tal manera que toda transición entre estados físicos de S refleje una transición en C. Dicho de otra manera, para que S implemente la computación C debe ocurrir que si hay una transición del estado c_0 al estado c_1 en C, entonces el sistema físico S se moverá del estado s_0 que representa c_0 al s_1 que representa c_1 (en alguna escala de tiempo establecida).

Esta definición de implementación es simple e intuitiva, lo cual es una ventaja grande para analizarla. Una consecuencia directa es el teorema que Putnam demuestra, que dice que cualquier sistema físico con suficientes estados (por ejemplo, una piedra al sol) implementa cualquier computación, trivializando el concepto (en lo que luego se dio en llamar pan-computacionalismo (Piccinini, 2010) La demostración es simple y no la repetiremos acá. Otros intentos de complejizar los requisitos del sistema para indicar que computa siguen, de alguna manera, dentro de la misma lógica que intentaremos cambiar: centrar la noción de computación meramente en los cambios de estado, olvidando que computar es realizar una serie de operaciones mecánicas dadas por reglas explícitas. En el sistema de Putnam esas reglas no aparecen en ningún lado, es decir, pueden inferirse o presuponerse, pero estarían sólo en un nivel abstracto, no tendrían ninguna existencia concreta en el sistema mismo.

Queda claro, entonces, en esta descripción el énfasis puesto en la computación vista como cambio de estados. El teorema de Putnam muestra que de esta manera todo (o casi) computa, lo cual hace perder especificidad a la idea de computación. Varias propuestas intentan buscar características más restrictivas para eliminar los ejemplos triviales (contrafácticos causales que los sistemas deben satisfacer, descripciones disposicionales, caracterizaciones mecanicistas) Encontrar el conjunto de propiedades que efectivamente permita distinguir los sistemas computacionales es interesante en sí, ya que ayuda a entender alcance y límites de la noción de computación. Creemos, sin embargo, que la característica distintiva que vuelve a un sistema computacional (en un cierto grado, también podremos distinguir de manera gradual si un sistema es más o menos computacional) pasa por que este sea programable, es decir, que puede inducirse al sistema a comportarse de maneras diversas a partir de proveerle una codificación (programa) que prescriba dichos comportamientos. Este programa será una realización de un objeto abstracto (al menos puede pensarse como tal) pero la forma de la codificación será dependiente del sistema en cuestión.

De alguna manera, tanto el análisis de Turing como el trabajo posterior de Gandy muestran qué significa que un sistema (sea físico o no) implemente un objeto abstracto determinado por un algoritmo o programa. La capacidad de realizar iterativamente pasos simbólicos elementales establece un vínculo entre las prescripciones (programas) y la realización efectiva de un comportamiento. El comportamiento puede ser considerado tanto en su versión abstracta (por ejemplo el grafo de una función numérica) como física (la salida en pantalla de un valor para cada entrada que se ponga). Aunque es menos obvio, también el programa puede ser considerado en sus dos "modos de existencia". El vínculo entre programa abstracto y comportamiento abstracto es lo que debe ser implementado en un sistema físico. Suele olvidarse la realización física del programa como componente esencial de una implementación de un sistema computacional. Esta puede tener diversas formas, por ejemplo un texto codificado en la memoria de una computadora, pero también una distribución determinada de posiciones de interruptores o la escritura de una tabla de transición para una máquina de Turing dada a ser leída por el "computor". La implementación o realización de un sistema computacional es entonces una relación entre relaciones. La relación "concreta" se realiza de maneras diversas de acuerdo a la tecnología empleada, y la justificación de su adecuación es en general empírica.



La relación que se establece entre el programa abstracto y el proceso abstracto dada por el intérprete, es una relación matemática que puede pensarse también de manera normativa. Los trabajos de la primera mitad del siglo XX que desarrollaron el concepto de computabilidad efectiva parecen referirse sobre todo a una caracterización de esta relación.

Una implementación de un sistema computacional descripto por un intérprete dado, consiste en una manera de realizar el programa y una manera de observar los comportamientos del sistema cuando se le provee dicho programa. La manera en que se observa el sistema permite definir qué será un proceso en ese sistema. El criterio de

corrección de la implementación es que ese proceso sea una realización del proceso abstracto asociado al programa por el intérprete. En este sentido, puede decirse que lo que se está haciendo es implementar el intérprete mismo. Poder mostrar que la relación establecida en el sistema físico refleja la relación abstracta puede requerir tanto métodos matemáticos como empíricos, y dependerá de la tecnología usada para establecer la realización.

La distinción abstracto-físico planteada en el esquema, puede pensarse como un a distinción analítica, al menos cuando se trabaja con sistemas realizados físicamente. En algún sentido puede hablarse de cierta preeminencia del plano superior, ya que no solo es objeto de estudio per se, sino que es necesario para que cobre sentido la relación del plano inferior. Funciona como condición prescriptiva y como criterio para determinar el grado en el que un sistema es computacional.

El esquema permite también ubicar los diferentes intentos de resolver el problema del pan-computacionalismo, o, más precisamente, contextualizar que significan en cada caso. El teorema de Putnam puede verse a partir de la elisión de la esquina inferior izquierda (como casi todas las versiones de pan-computacionalismo). Si no es necesaria una realización física del programa mismo, es decir, solo la computación se realiza físicamente, el grado de libertad que se gana claramente trivializa la noción de lo físicamente computable.

El intento de Copeland (Copeland, 1996) de caracterizar la noción de realización de un sistema a partir de la de modelo en lógica termina colapsando horizontalmente el dibujo, dada la necesidad de homogeneizar en un mismo lenguaje el programa y el comportamiento. Para Copeland el comportamiento se obtiene a partir del programa y de reglas de acción que se infieren de las instrucciones interpretadas (lo que llama algoritmo+arquitectura). La dimensión perdida hace que aparezcan modelos "indeseables" teniendo que recurrir a propiedades extra-lógicas para separar los modelos adecuados de los que no, poniendo en duda la utilidad de la caracterización misma.

Conclusiones

Recuperar las dimensiones analíticas perdidas en la mayor parte de los trabajos acerca de qué significa que un sistema físico compute parece una prometedora manera de resolver varios problemas filosóficos extrañamente persistentes y ubicuos, mayormente relacionados con el (pan-) computacionalismo. Argumentamos, brevemente aquí, que esas dimensiones estuvieron siempre presentes en los trabajos fundacionales el área, que incluso no se pierden en trabajos posteriores como los de Gandy y Sieg (Sieg and Byrnes, 1996) (ciertamente quedan implícitas en Gandy, dado el énfasis diferente del trabajo). Los intentos de determinar las propiedades que hacen que un sistema sea computacional, no usan de manera determinante el concepto de programa. Pensamos que esto puede deberse a que no hay una caracterización ontológica directa de qué es un programa. Sin embargo, la noción aquí presentada permite una definición relacional simple cuya utilidad está a la vista.

Notas

[i] Cabe señalar que los problemas planteados por Wittgenstein y analizados por Kripke acerca de qué significa seguir una regla no se aplican a los pasos elementales de la máquina Turing. Un conjunto finito de acciones posibles sobre un alfabeto finito puede describirse exhaustivamente, no hace falta una comprensión o un acuerdo que vaya más allá de una mera enumeración. La aplicación repetida de

pasos elementales no es lo que estaba en el foco de crítica de Kripke. Quizá esto pueda explicar la aceptación que produjo el trabajo de Turing por sobre los intentos previos de Church, Gödel, Herbrand, etc

Bibliografía

ABELSON, Harold; SUSSMAN, Gerald J. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, MA: MIT Press, 1996.

BLANCO, Javier; CHERINI, Renato; DILLER, Martin; GARCIA, Pío. Intepreters as computational mechanisms. Pp. 52-55, en MAINZER, Klaus (ed.) *Proceedings of 8th Conference on Computing and Philosophy*, 2010.

BLANCO, Javier; CHERINI, Renato; DILLER, Martin, GARCIA, Pío. A behavioral characterization of computational systems. Pp. 32-36, en ESS, Charles, HAGENGRUBER, Ruth (eds.) *Proceedings of 1st Conference of the International Association on Computing and Philosophy*, 2011.

COPELAND, B. Jack. What is computation? *Synthese* **108** (3):335-59, 1996

DIJKSTRA, Edsger W. *On the cruelty of really teaching computing science*. Circulación privada, 1988.

EDEN, Amnon H. Three paradigms of computer science *Minds and Machines* **17** (2):135–167, 2007

EDEN, Amnon H., TURNER, Raymond. The Philosophy of Computer Science. En ZALTA, Edward N. (eds.). *The Stanford Encyclopedia of Philosophy*. 2011.

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

HE, Jifeng; HOARE C. A. R. Unifying theories of programming. Pp. 97-99, en ORLOWSKA, Ewa, SZALAS, Andrzej (eds.). *RelMiCS*. 1998.

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

PICCININI, Gualtiero. Computation in Physical Systems. En ZALTA, Edward N. (eds.). *The Stanford Encyclopedia of Philosophy*. 2010.

PUTNAM Hilary. *Representation and Reality*. Cambridge, MA: MIT Press, 1987.

SEARLE, John. Is the brain a digital computer? Pp. 21-37, en *Proceedings and Addresses of the American Philosophical Association*, 1990.

SIEG, Wilfried, BYRNES, John. K-Graph Machines: Generalizing Turing's Machines and Arguments. Pp. 98-119, en HAJEK, Petr (ed.). *Proceedings of Gödel'96: Logical foundations of mathematics, computer sicence and physics – Kurt Gödel's legacy*, 1996.

TURING, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematics Society* series 2, 42 (1): 230–265, 1937.