EPISTEMOLOGÍA E HISTORIA DE LA CIENCIA

SELECCIÓN DE TRABAJOS DE LAS XX JORNADAS
VOLUMEN 16 (2010)

Pío García
Alba Massolo

Editores

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

Esta obra está bajo una Licencia Creative Commons atribución NoComercial-SinDerivadas 2.5 Argentina

La noción de intérprete y la naturaleza de la ciencia computacional

*Jaime Blanco**

Introducción

La ciencia de la computación tiene su origen en las consecuencias del "fracaso" del programa de Hilbert. La idea de proceso efectivo se incorporó como uno de los temas centrales de la pujante nueva disciplina de la metamatemática. No parece casualidad que las primeras computadoras fueran creadas por equipos de trabajo dirigidos por matemáticos que trabajaban (entre otras cosas) en estos temas (ejemplos paradigmáticos de esto son Alan Turing y John Von Neumann). La aparición de las computadoras dio lugar a líneas de investigación interconectadas entre lógica, matemática e ingeniería, las cuales, a partir de encontrar problemas específicos y nuevos métodos de solución, van constituyendo una nueva disciplina. Las nociones de programa y de algoritmo son centrales en esta nueva ciencia, y sus diversas caracterizaciones como fórmulas lógicas, artefactos o procesos da lugar a concepciones diferentes tanto ontológicas como epistemológicas y metodológicas. Desde estas diferentes concepciones, la ciencia de la computación ha sido planteada como ciencia formal, como ciencia empírica o directamente como una disciplina tecnológica. Queda la pregunta de si estas miradas presentadas como paradigmas diferentes (Eden) no pueden ser integradas como enfoques compatibles de distintos aspectos de la disciplina. La noción introducida por Chaitin de ciencia quasi-empírica (para el caso de la matemática) puede ser relevante para caracterizar al menos ciertas áreas de la ciencia de la computación las cuales tienen puntos en común con las ciencias empíricas pero también diferencias importante que varios trabajos en el área pasan por alto.

Ciertas novedades importantes aparecen en la ciencia de la computación, las cuales deberían ser consideradas seriamente en cualquier intento explicativo de la naturaleza de esta ciencia. La siguiente cita de Dijkstra (del artículo On the cruelty of really teaching computing science), ilustra uno de los puntos más novedosos, la inigualable complejidad de niveles de abstracción presentes en cada programa y la necesidad de contar con herramientas para manejarlos.

> "The first radical novelty is a direct consequence of the raw power of today's computing equipment. We all know how we cope with some- thing big and complex; divide and rule, i.e. we view the whole as a compositum of parts and deal with the parts separately. And if a part is too big, we repeat the procedure. The town is made up from neighbourhoods, which are structured by streets, which contain buildings, which are made from walls

* Universidad Nacional de Cordoba

> and floors, that are built from bricks, etc. eventually down to the elementary particles. And we have all our specialists along the line, from the town planner, via the architect to the solid state physicist and further. Because, in a sense, the whole is "bigger" than its parts, the depth of a hierarchical decomposition is some sort of logarithm of the ratio of the "sizes" of the whole and the ultimate smallest parts. From a bit to a few hundred megabytes, from a microsecond to a half an hour of computing confronts us with completely baffling ratio of 10?! The programmer is in the unique position that his is the only discipline and profession in which such a gigantic ratio, which totally baffles our imagination, has to be bridged by a single technology. He has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before. Compared to that number of semantic levels, the average mathematical theory is almost flat. By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history."

¿Cómo es posible dar cuenta de este desafío? El mismo Dijkstra y muchos de sus seguidores (Gries, Feijen, van Gasteren, Backhouse, Kaldewaij, etc.) han consistentemente construido un marco conceptual y práctico en el cual hacerlo. Sus ideas se basan en el uso del razonamiento lógico y matemático en cada paso del desarrollo de programas, obteniendo no solo una garantía de corrección, sino también herramientas metodológicas que colaboran en el procesos de desarrollo de los programas mismos.

Pero esto es un solo lado de la historia, digamos el lado humano. ¿Cómo se refleja esta novedad en la estructura misma de los artefactos que se crean? Algunas respuestas a esta cuestión están apareciendo en la literatura reciente de filosofía de las ciencias computacionales. Por ejemplo, la noción de implementación desarrollada por Rapaport y los estudios sobre como funciona la abstracción en ciencia de la computación Esta idea también aparece en desarrollos tecnológicos concretos. Lo tipos abstractos de datos son uno de los ejemplos más estudiados, al igual que la teoría de la implementación de lenguajes de programación de alto nivel, la cual provee herramientas invaluables para cerrar la brecha: compiladores, intérpretes y combinaciones de ambos.

En este trabajo, sugerimos que la noción de intérprete puede ser vista como concepto clave para comprender aspectos específicos de ciencia de la computación a la vez que permite formular problemas filosóficos. Mientras que en la literatura técnica los intérpretes y compiladores son analizados usualmente de acuerdo a su performance, creemos que desde un punto de vista conceptual la noción de intérprete es más fundamental e inevitable para comprender los sistemas computacionales.

Otro camino conceptual, completamente diferente, considera también a esta noción como central. Nos referimos a la teoría de la computación surgida a partir de los trabajos de Turing,

Church, Post, Kleene, etc. desde la década de 1930 en adelante. La llamada hoy teoría de la computación intenta explicar la noción de procedimiento efectivo, sus alcances y sus límites. Dicha teoría tiene una amplia influencia en problemas de filosofía de la mente y de la matemática.

Gregory Chaitin analiza, en términos de teoría de la información, la continuidad conceptual entre los trabajos de Gödel y Turing, los cuales muestran respectivamente la incompletitud inherente de ciertas axiomatizaciones y la existencia de problemas incomputables. Las construcciones usadas para ambos teoremas se basan en ciertas posibilidades reflexivas de ambas teorías, las cuales permiten "codificar" como datos algunas propiedades meta-teóricas esenciales. En el caso de las presentaciones de lógica de primer orden para la aritmética con la que trabaja Gödel, dicha codificación consiste en asociar a cada fórmula misma un número el cual puede en principio funcionar como argumento de una fórmula, en el caso de las máquinas de Turing, puede codificarse cualquier máquina como dato de entrada de otra. Esto permite en el primer caso, internalizar propiedades metamatemáticas como formulas de la teoría, y en el segundo caso, a través de la máquina universal, disponer de un mecanismo que puede comportarse como cualquier otro cuya codificación toma como entrada. Esta propiedad permite pensar a la máquina universal de Turing como un mecanismo *programable*.

Tanto en su fundación histórica como teórica y práctica, la ciencia de la computación descansa en la idea de mecanismo programable, es decir, que se dispone de una sintaxis a partir de la cual pueden codificarse una gran variedad de comportamientos que dicho mecanismo puede llevar a cabo. Si bien existe una diferencia ontológica clara entre mecanismos físicos y teorías formales como las máquinas de Turing o el lambda cálculo, con respecto a las propiedades relevantes para caracterizar procedimientos efectivos, la propiedad de ser programable puede definirse de manera análoga para ambos tipos. La relación entre mecanismos físicos y descripciones formales puede ser descriptiva (una caracterización de como pensar comportamientos causales de manera computacional) o prescriptiva (una manera de realización sistemática de descripciones formales que satisfagan criterios de adecuación).

Intérpretes

Sea S un conjunto de comportamientos, L una función de codificación que tiene a L como imagen. Un intérprete I para la codificación L el cual implementa los comportamientos de S es un mecanismo que toma como entrada una codificación P y ejecuta el comportamiento asociado, es decir L(P). En caso que el comportamiento requiera datos de entrada, el mecanismo tendrá alguna manera de incorporarlos a partir de una codificación M propuesta.

Entendemos por mecanismo, siguiendo a Piccinini, a un sistema sujeto a una explicación en términos de la organización y funciones de los componentes. No es necesario en nuestra descripción hacer referencia a los estados internos de los mecanismos.

No se fijará una única noción de comportamiento. Dependiendo de las propiedades de interés, pueden considerarse diferentes comportamientos: descripciones de entrada-salida, conjuntos de trazas de ejecución, grafos de procesos, conjuntos de entrelazamientos (interleavings) de acciones atómicas, etc. Para fijar los conceptos, nos concentraremos en la descripción de entrada-salida (input-output) de los comportamientos, la cual es la más estudiada y una de las más útiles. Un intérprete satisface entonces la siguiente ecuación.

$I(P,M(D)) = L(P)(D)$

Usualmente se llama *programa* a la codificación P y la función L será una descripción del lenguaje de programación asociado. El comportamiento resultante es la computación.

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. Ilustramos eso a partir de los siguientes ejemplos:

• 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 cuyos datos de entrada están en una cinta, y va aplicando mecánicamente los pasos descriptos en esa tabla. Cada pasos indican posibles modificaciones a una posición de la cinta y un posible cambio de posición y cual es la siguiente instrucción. Si no hay siguiente, el programa termina. Un conjunto de comportamientos que este intérprete implementa es el de las funciones computables. La función de codificación de los comportamientos esta dada por la (en general difícil) codificación de una función en los pasos elementales de este formalismo.

• 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. La función de codificación L es la que expresa esas diferentes miradas.

• El hardware de una computadora 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.

• Una computadora programable por hardware (es decir, cambiando cables o switchs de manera física) como por ejemplo la ENIAC. Los comportamientos implementados son los mismos que en el caso anterior. En este caso, la codificación de los programas y de los datos están dados por una combinación de configuraciones del hardware y 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 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 cálculo lambda tipado (simple) es un intérprete para un conjunto no-trivial de funciones totales. Los comportamientos se codifican como términos lambda al igual que los datos de entrada.

• Dada una semántica operational determinística y computable para un lenguaje imperativo dado, alguien puede calcular (demostrar) la secuencia de los valores de las variables del programa a partir de la entrada hasta su terminación. Los comportamientos pueden estar dados aquí por las trazas de ejecución del programa (o, abstrayendo, solo considerar el valor final). Un demostrador de teoremas para una presentación lógica de dicha semántica podría también ser una realización de este intérprete. Este ejemplo se condice con la cita tomada de Richard Bornat:

> "A working program is a constructive proof that some computation is possible in a particular formal system."

Estos ejemplos revelan tanto los aspectos abstractos como concretos de los intérpretes. La descripción de las propiedades abstractas permite identificar las características esenciales de un sistema computacional, mientras que en la realización concreta descansa la idea de mecanismo efectivo. Más aún, un intérprete puede ser un programa considerado en un sentido tradicional, pero también el hardware de una máquina. La siguiente cita del libro de computabilidad de Jones, ilustra esto"

> The "computing agent" then interprets the algorithm, it can be a piece of hardware, or it can be software. an interpreter program written in a lower-level programming language. Operationally, an interpreter maintains a pointer to the current instruction within the algorithm's instruction set, together with a representation of that algorithm's current storage state. Larger algorithms correspond to larger interpreted programs, but the interpreter itself remains fixed, either as a machine or as a program.

La noción de intérprete considerada es funcional (podría pensarse como conductista, aunque en este contexto no parece una diferencia importante), esto es, un intérprete es tal cuando es capaz de producir comportamientos específicos a partir de programas con una codificación dada. Un programa es entonces una entidad sintáctica susceptible de ser interpretada. Un programa

es tal solamente relativo a un intérprete dado, y un intérprete es tal sólo para una codificación particular de un conjunto de comportamientos (un lenguaje de programación). Los conceptos de programa, lenguaje e intérprete son entonces inter-definibles.

Jerarquías de los sistemas de cómputo.

En esta sección describimos maneras naturales de evaluar la potencia computacional de un mecanismo. (Los detalles técnicos pueden encontrarse en el artículo de 2010 de Blanco, García, Diller y Cherini)

Esto permite reformular preguntas acerca de que significa que un mecanismo físico compute, y preguntarse en cambio cuanto computa, que sintaxis habilita que comportamientos. Los argumentos de Putnam y Searle que trivializan lo computable demostrando que cualquier objeto físico de alguna complejidad computa cualquier máquina de Turing, se pueden reubicar a la luz de esta clasificación, dado que para ninguna codificación adecuada la pared o la piedra pueden comportarse de diversas maneras interesantes. En otras palabras, son intérpretes muy pobres.

Es necesario acá aclarar que a diferencia de los estados internos de un sistema computacional, los símbolos de entrada y salida deben estar preestablecidos, es decir, las codificaciones sintácticas son un parámetro del sistema computacional. Esto parece ser aceptado por el mismo Putnam, quien sin embargo argumenta que si se tienen las relaciones de entrada-salida adecuada, cualquier programa que las satisfaga estaría implementado por ese sistema, reduciendo el funcionalismo al conductismo. Si bien para le caso de una mente esto podría tener consecuencias indeseables, para caracterizar ontológicamente a los programas y consecuentemente a las ciencias de la computación, no parece ser esto un problema, es más, podría considerarse que es el nivel de abstracción adecuado. No distinguir una Mac de una PC consideradas como sistemas computacionales no parece contradecir la teoría ni las prácticas de la ciencia computacional. Veremos que la propiedad de ser un intérprete de un lenguaje rico (por ejemplo un lenguaje equivalente a las máquinas de Turing) solo es satisfecha por sistemas que pueden sin problema ser considerados computacionales.

Pueden pensarse dos nociones adecuadas para definir una jerarquía (un pre-orden) de sistemas computacionales (con sus respectivas funciones de codificación)

La más simple y que de alguna manera coincide con varias nociones jerárquicas (por ejemplo con la jerarquía de Chomsky), es el de la inclusión del conjunto de comportamientos. En dicho pre-orden, un sistema A es más computacional que otro B si A puede hacer todo lo que hace B (y posiblemente más cosas). Esta noción coincide con la de existencia de un compilador del lenguaje de B en el lenguaje de A (no necesariamente escrito en el lenguaje de ninguno de ellos).

Una noción menos común, más restrictiva pero filosóficamente quizá más relevante, es la existencia de un intérprete. Un sistema A es computacionalmente más rico que B si existe un intérprete pare el lenguaje de B escrito en el lenguaje de A.

De alguna manera, la primera noción de orden corresponde con la inclusión de comportamientos, mientras que la segunda con la pertenencia, es decir que B mismo es un comportamiento posible de A.

Ambas nociones son transitivas, dado que tanto componer un compilador con un intérprete como dos intérpretes entre si (obviamente con el dominio y co-dominio adecuados) da como resultado otro intérprete.

A partir de disponer de estas jerarquías, puede hablarse de grados de computación de un sistema dado, lo cual se condice con las diferentes potencialidades de diversos artefactos en cuanto a su programabilidad.

Scipts, procesos y el debate acerca de la verificación de programas.

En su artículo, Eden distingue dos nociones diferentes de programa

> "We seek to distinguish between two fundamentally distinct senses of the term program in conventional usage: The first is that of a static script, namely a well-formed sequence of symbols in a programming language, to which we shall refer as a program-script. The second sense is that of a process of computation generated by executing a particular program-script, to which we shall refer as a program-process."

Esta noción dual de la naturaleza de los programas es clave para plantear y responder varias preguntas centrales acerca de la ciencia de la computación. Claramente el problema de establecer la relación o vínculo entre las nociones de script y proceso es relevante filosóficamente y atravesado por diversas controversias. En el artículo citado, la relación entre ambas nociones se usa fuertemente para establecer si la ciencia de la computación es una ciencia formal, una ciencia empírica o una ingeniería. Sorprendentemente, la noción de intérprete como vínculo entre ambas maneras de considerar a los programas no fue nunca considerada.

La brecha ontológica introducida en la presentación de Eden tiene consecuencias teóricas y prácticas. Las teóricas son claras: cómo relacionar las entidades de diferentes lados de la brecha pasa a ser el problema de la implementación. La relación entre el conocimiento acerca de un script y el de su respectivo proceso queda oscura. Una de las preguntas prácticas más importantes que se siguen, es qué tipo de confianza podemos tener en la verificación de los programas, o cual es la justificación que puede proveer el testing.

Gran parte del debate acerca de la verificación formal está atravesado por esta relación, la cual no ha sido debidamente explicada. Algunos autores sostienen que dado que los procesos no son

expresiones matemáticas, no serían susceptibles de ser analizados matemáticamente, perdiendo de vista que han sido generados por una expresión matemática. Esta omisión suele llevar a un análisis de los procesos que solo considera una forma particular de estos, en general procesos físicos generados por máquinas con una arquitectura Von Neumann. Estos procesos son considerados por Eden como entidades temporales, no-físicas, causales, metabólicas, contingentes respecto de una manifestación física y no-lineales.

En un intento por explicar esta relación, nos encontramos con que el concepto de intérprete, considerado en toda su generalidad, estuvo siempre disponible para ser considerado como vínculo entre las dos manifestaciones de los programas. Su definición usual en ciencias de la computación postula exactamente eso, que un intérprete es un programa que toma el código fuente (script) de otro programa y lo ejecuta. Una generalización natural y que ocurre en la literatura, es considerar otros mecanismos (no necesariamente un programa) como intérpretes, si realizan la misma función.

Tomar a los intérpretes como vínculo relativiza la caracterización de los procesos propuesta por Eden, dado que estas características no son universales sino contingentes respecto de ciertas formas tecnológicas. La existencia de intérprete funciona como criterio para determinar cuando un determinado proceso esta asociado a un script y dicho criterio es aplicable a diversas arquitecturas.

En este contexto el problema de la verificación formal queda mejor planteado, dado que un proceso queda caracterizado en términos de propiedades de alto nivel las cuales son invariantes respecto de la "traducción" realizada por este. Las propiedades de los scripts pueden ser demostradas usando la descripción abstracta del intérprete y cierto andamiaje formal. Cómo un sistema físico realiza el intérprete se establecería entonces usando la teoría física y podría ser validado empíricamente. Esta presentación es consistente con la deflación ontológica propuesta en el trabajo de 2008 de Blanco y García, en la cual tratamos de disolver ciertas dificultades considerando tanto aspectos prescriptivos como descriptivos de los sistemas computacionales.

Pan-computacionalismo

La idea de que cualquier sistema físico implementa cualquier autómata finito (o realiza cualquier función en alguna otra de sus versiones), parece implicar una trivialización de la noción de computación, ya que si todo computa, nada específico habría en ello.

No hace falta entrar en detalles aquí acerca de las construcciones con las cuales se justifica que una piedra o un balde de agua o una pared puede verse como computando cualquier programa. El hecho básico es que si se tienen suficientes estados diferentes entre sí, puede interpretarse a estos como un secuencia de estados de un programa cualquiera. Múltiples discusiones y refutaciones

has surgido a este "teorema", pero esencialmente está basado en la apreciación correcta de que los estados computacionales internos son no tienen ninguna estructura intrínseca, que pueden ser implementados de infinidad de maneras.

Diferente suerte tienen las formas de codificar la entrada-salida. Dado que de esta manera los sistemas computacionales se comunican con el resto del mundo, la forma de codificación debe estar preestablecida, de hecho suele estar dada por secuencias cuyos elementos pertenecen a algún conjunto finito. En este sentido, se conforma con una versión menos fuerte de pan-computacionalismo, ya que afirma que cualquier sistema con las relaciones de entrada salida adecuada implementa cualquier programa que se condiga con dicho comportamiento observable. Se reduciría el funcionalismo al conductismo.

Pensar a los sistemas computacionales como implementando intérpretes, permite postular argumentos simples en contra del pan-computacionalismo, los cuales reformulan el problema.

Pensar que un sistema es computacional si se lo puede ver como un intérprete implica per se grados de computacionalidad (dependiendo de los comportamientos que pueden codificarse para dicho intérprete). En este sentido, no habría sistemas intrínsecamente computacionales sino con un cierto grado de programabilidad. Esto parece volver a poner el énfasis no en que una computación efectivamente ocurra, sino en el potencial de producir comportamientos diversos a partir de una sintaxis dada.

Siguiendo esta idea, puede verse que en el argumento de Searle reconstruido por Copeland, se muestra que para todo sistema físico F (con un suficiente número de partes discriminables) y para toda especificación S de un programa existe una función de etiquetado L (determina la relación entre el sistema físico y el algoritmo especificado por S) tal que (F,L) es un modelo de S (en sentido lógico, S es una descripción en lógica del programa y de la arquitectura donde corre).

Ahora bien, la especificación S que Copeland presenta consiste en la ejecución de un programa particular (con sus datos de entrada incorporados). Se podría decir que con esto alcanza, dado que para cualquier programa se puede encontrar el etiquetado L necesario para que sea ejecutado en ese sistema físico. Sin embargo, que un sistema físico implemente un intérprete significa que para cualquier codificación de un programa en su entrada el sistema debe comportarse como se requiere con *una única codificación,* siempre la misma para cualquier programa de entrada. Este intercambio de cuantificadores hace que la realización de un intérprete sea mucho más difícil de obtener. Esto ocurre porque la propiedad de ser un intérprete parece ser de "alto orden", es decir que no se reduce a casos particulares.

Respecto de la extensión que hace Putnam para sistemas con entrada-salida, la falla en ese argumento queda de manifiesto cuando se intenta ver que significa implementar un intérprete. El teorema requiere que el sistema computacional satisfaga la relación de entrada salida pertinente,

y entonces los estados internos no importarían. En el caso de un intérprete, satisfacer la relación de input-output es un requisito difícil de cumplir, es decir, que solo los mecanismos efectivamente programables lo satisfarían (para cada codificación de un programa de entrada, debe comportarse como tal, lo cual solo un sistema con gran poder de cómputo puede hacer).

Conclusiones

Propusimos en este trabajo una noción levemente generalizada de intérprete como herramienta explicativa para cuestiones centrales de ciencia de la computación, tanto en sus fundamentos teóricos como en sus prácticas.

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

- Se establecen condiciones mínimas para que un sistema sea computacional para 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 en su artículo de 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, etc.)

- Dado que la composición de intérpretes o aún 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 diferente niveles.

- Algunas cuestiones ontológicas (¿qué es un programa?, ¿cuándo es computatcional un sistema?, etc.) 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. El grado de computación de diferentes sistemas puede ahora ser comparado.

- Permite aclarar y presentar nuevos argumentos en el debate acerca de si la noción de computación requiere una aproximación semántica o no. Un intérprete puede ser visto como una generalización de la noción de Piccinini de semántica interna. Algunas relaciones verdaderamente semánticas puede ser registradas en la descripción de los comportamientos deseados y en la descodificación usada para "leer" el comportamiento resultante producido por el intérprete.

La noción de intérprete usualmente usada en ciencias de la computación adquiere una importancia explicativa central cuando se la entiende en términos filosóficos (algo que de hecho Chaitin explota en varios de sus resultados). Permite presentar varias cuestiones de ciencias de la computación de manera precisa y en varios niveles de abstracción. Por último, da lugar a una deflación ontológica acerca de las diferencias entre máquinas físicas y abstractas consistente con las prácticas y que habilita una discusión mucho más fértil acerca de diferentes aspectos prescriptivos y descriptivos de la ciencia de la computación.

Referencias Bibliográficas

Javier Blanco and Pio Garcia. A categorial mistake in the formal veri- fication debate. In European Conference on Computing and Philosophy (ECAP), June 2008.

Javier Blanco, Pio García, Martin Diller y Renato Cherini. Interpreters: towards a philosophical account of computer science. En proceso de evaluación para Minds & Machines, 2010

Richard Bornat. Is computer science science? In European Conference on Computing and Philosophy (ECAP), July 2006.

Gregory Chaitin. Algorithmic information theory. Cambridge University Press, 2004.

Gregory Chaitin. From Philosophy to Program Size Tallinn Institute of Cybernetics, 2003

David Chalmers. Does a rock implement every finite-state automaton, 1996.

B. Jack Copeland. What is computation? Synthese, 108(3).335–59, 1996

Edsger W. Dijkstra. A Discipline of Programming. Prentice Hall, 1976. Edsger W. Dijkstra. On the cruelty of really teaching computing science. circulated privately, December 1988.

Amnon H. Eden. Three paradigms of computer science. Minds Mach., 17(2):135–167, 2007.

Amnon H. Eden and Raymond Turner. Problems in the ontology of computer programs. Applied Ontology, 2(1).13–36, 2007

Robin Gandy. Church's thesis and principles for mechanisms. In K. J. Barwise, H. J. Keisler, and K. Kunen, editors, The Kleene Symposium, vol- ume 101, pages 123–148, 1978.

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

John McCarthy. Towards a mathematical science of computation. In Proceedings of IFIP Congress, pages 21–28, North-Holland, 1962.

Davis Martin and Weyuker Elaine J. Computability, complexity, and languages . fundamentals of theoretical computer science. Academic Press, New York, 1983

James H. Moor. Three myths of computer science. British Journal for the Philosophy of Science, 29(3):213–222, 1978.

Gualtiero Piccinini. Computation without representation. Philosophical Studies, 137(2), 2008.

Gualtiero Piccinini. Computers. Pacific Philosophical Quarterly, 89(1).32–73, 2008.

Hilary Putnam. Representation and Reality. MIT Press, Cambridge, MA, USA, 1988.

William J. Rapaport. Implementation is semantic interpretation. Monist, 82:109–130, 1999

John Searle. Is the brain a digital computer?, 2004.

Raymond Turner. Understanding programming languages. Minds Mach., 17(2).203–216, 2007