Ciencia de la computación y filosofía: unidades de análisis del software

Principia: An International Journal of Epistemology, 22(2): 203–227

Juan M. Durán*

High Performance Computing Center Stuttgart
University of Stuttgart

Abstract

Una imagen muy generalizada a la hora de entender el software de computador es la que lo representa como una "caja negra": no importa realmente saber qué partes lo componen internamente, sino qué resultados se obtienen de él según ciertos valores de entrada. Al hacer esto, muchos problemas filosóficos son ocultados, negados o simplemente mal entendidos. Este artículo discute tres unidades de análisis del software de computador, esto es, las *especificaciones*, los *algoritmos* y los *procesos computacionales*. El objetivo central es entender las prácticas científicas e ingenieriles detrás de cada unidad de software, así como analizar su metodología, ontología y epistemología.

There is a widely extended image of computer software as some sort of 'black box,' where it does not matter how it internally works, but rather what sort of results are obtained given certain input values. By approaching computer software this way, many philosophical issues are hidden, neglected, or

*duran@hlrs.de - juanduran@gmail.com

simply misunderstood. This article discusses three units of analysis of computer software, namely, *specifications*, *algorithms*, and *computer processes*. The aim is to understand the scientific and engineering practices supporting each unit of analysis, as well as to analyze their methodology, ontology, and epistemology.

Keywords— Especificación, Algoritmo, Proceso computacional, Ciencia de la computación, Filosofía de la computación

Specification, Algorithm, Computational Process, Computer science, Philosophy of computer science

1 Introducción

Una imagen muy generalizada a la hora de entender el software de computador es la que lo representa como una "caja negra": no importa realmente saber qué partes lo componen internamente, sino qué resultados se obtienen de él según ciertos valores de entrada. En este sentido, la habilidad que tiene el software de computador para proporcionar información sobre el mundo –por ejemplo, en el contexto de la práctica científica e ingenieril– se ve plasmada en su capacidad de producir resultados confiables sobre el mundo. Estas ideas tienen su origen en una evaluación y análisis reduccionista de lo que consiste el software de computador. En particular, no hace justicia a los procesos de diseño, programación y justificación del software de computador. De aquí que sea importante entender que el software de computador depende de procesos muy complejos y creativos que requieren de una comprensión más acabada de su naturaleza.

En este artículo presento y discuto tres unidades del software de computador. Estas son: las *especificaciones*, los *algoritmos* y los *procesos* asociados a la computación.[1] Asimismo, discuto una serie de problemas filosóficos asociados a cada una de estas unidades del software, así como la relación que tienen entre sí. Uno de los objetivos principales de este artículo es profundizar en la comprensión de las prácticas que hay detrás de cada unidad de software, cómo son construidas, programadas, justificadas y utilizadas. Para lograr este fin, combino los aspectos técnicos de cada unidad con problemas filosóficos específicos.

A pesar de que este artículo es un estudio general, su importancia reside en el hecho

que visibiliza la naturaleza del software y los problemas de corte filosófico que este acarrea, temas que son usualmente ignorados por la comunidad científica e ingenieril. De hecho, un objetivo del artículo es trazar un mapa que posibilite identificar y ubicar correctamente problemas filosóficos relacionados con cada unidad de análisis del software. Las malas interpretaciones y los errores a la hora de identificar correctamente estos problemas filosóficos han llevado a investigadores de distintas áreas a no poder ubicar el origen y la naturaleza de los problemas técnicos, con consecuencias relevantes para sus propias disciplinas. Un ejemplo de esto son las preguntas sobre la confianza que tenemos en el software (e.g., el llamado 'argumento de opacidad epistémica'). Aunque tradicionalmente se ha ubicado a la opacidad en el nivel del proceso que lleva adelante el cómputo (Humphreys, 2004), puede ser igualmente interpretado como localizado al nivel de la especificación (Newman, 2015). Por estas razones, este artículo pone la atención en problemas metodológicos y epistemológicos de la especificación del software; en problemas ontológicos y epistemológicos asociados a la noción de algoritmo; y en cuestiones sobre confianza epistémica de los procesos de cómputo que vienen de la mano de estudios sobre verificación, validación y justificación de los resultados de computar.

La importancia de este estudio es que puede ser incluido dentro de los estudios que entienden que las computadoras son unidades de análisis filosófico y tecnológico en sí mismas. En este sentido, este artículo continúa la tradición iniciada por James H. Moor en su contribución de (1978), que desmantela tres mitos de la ciencia de la computación, a saber, software vs. hardware, digital vs. análogo, y modelo vs. teoría. El objetivo principal de Moor era argüir que muchas veces los investigadores tienen ideas erróneas sobre la ciencia de la computación y disciplinas relacionadas, y que, por lo tanto se crean mitos alrededor de estas. Algo similar ha argumentado Björn Schembera sobre mitos en simulaciones computacionales (Schembera, 2017). Este artículo, entonces, continúa con esa tradición de desmitificar, si se quiere, el software de computador. Para lograr este fin, haré uso tanto de la filosofía de la ciencia tradicional como de la filosofía de la ciencia de la computación y disciplinas relacionadas (e.g., software engineering, teoría de la computación, etc.).

La estructura de este trabajo es la siguiente: la sección 2 discute la noción de *especificación* en software de computador junto con preguntas filosóficas de importancia. La sección 3 repite el mismo formato pero lo hace sobre la noción de *algoritmos*. A diferencia de la sección anterior, aquí adiciono una discusión técnica y filosófica sobre la conexión de los algoritmos con las especificaciones. De manera similar, la sección 4 reconstruye la

noción de *proceso computacional* y la discute en conexión con la de algoritmos y especificaciones. Finalmente, la sección 5 presenta las conclusiones y propone futuras líneas de trabajo.

2 Especificaciones

A fin de comprender el estudio de las especificaciones en software de computador, establezcamos una analogía con instrumentos científicos como punto de partida. La validez de esta analogía tiene dos raíces. Por un lado, nos permite comprender el software de computador desde una perspectiva más familiar; por el otro, hace explícito un supuesto de las especificaciones del software de computador, a saber, que pertenecen, al menos en este nivel de análisis, a la misma categoría conceptual que las especificaciones de los instrumentos científicos. Esto queda en evidencia en el análisis metodológico y epistemológico de ambos tipos de especificaciones. Desde luego, mi análisis no niega la existencia de diferencias entre las especificaciones de instrumentos científicos y especificaciones de software de computador.

Cada instrumento científico requiere de la elucidación de su funcionalidad y operatividad, diseño y límites de uso. Para esto, se compila información que se conoce como la *especificación* del instrumento científico. Un ejemplo de esto es el termómetro de mercurio. Una posible especificación indica que el termómetro puede ser utilizado para medir la temperatura de líquidos, del ambiente, o la de una persona; no puede ser utilizado por encima de 50° C, o por debajo de -10° C; tiene un intervalo de 0.5° C entre la escala de valores y una precisión de $\pm 0.01°$ C. También se pueden dar otras especificaciones, a saber, que el mercurio en el termómetro se solidifica a -38.83° C. Otras especificaciones pueden también ser dadas, como que el mercurio debe ser introducido en la ampolla al final del termómetro; que el mercurio debe tener cierta cantidad de volumen, caso contrario las mediciones pueden ser imprecisas; que el resto del cuerpo del termómetro debe ser rellenado con nitrógeno, etc.

Además de estas especificaciones, usualmente se tiene información relevante sobre el sistema objeto. En el caso de medir la temperatura del agua, es relevante conocer que cambia de estado –de líquido a sólido a 0° C–; que si el agua no se aísla apropiadamente, la medición puede ser imprecisa dependiendo de si está en contacto con otras fuentes de temperatura; que la ley de la termodinámica de Zeroth explica lo que significa medir la

propiedad física 'temperatura', etc. Información como esta es fundamental para establecer la confianza que se tiene en el termómetro, así como en los resultados a partir de la medición de dicho termómetro. En otras palabras, que un termómetro funciona correctamente y que los valores obtenidos son confiables, quiere decir que la medición se llevó a cabo dentro dentro de las especificaciones provistas por el fabricante. Notemos que la situación inversa es también cierta, es decir, el uso del termómetro que explícitamente contradiga sus especificaciones y la información que tenemos del sistema objeto no puede garantizar mediciones confiables.

Richard Feynman nos da una anécdota interesante sobre las consecuencias del mal uso de un instrumento. Durante el tiempo en que fue miembro del comité sobre el desastre del Challenger, Feynmann recuerda la siguiente conversación con el fabricante de una pistola de infrarrojos:

> Señor, su pistola de infrarrojos no tuvo nada que ver con el accidente. Fue utilizada aquí de manera contraria a los procedimientos de su manual de instrucciones, yo sólo estoy tratando de entender si puedo reproducir el error y determinar cuáles fueron las temperaturas aquella mañana. Para hacer esto, necesito saber más sobre su instrumento. (Feynman, 2001, p. 165-166)

El final de esta historia es que la pistola de infrarrojos no tuvo nada que ver con el accidente del transbordador espacial Challenger. En cambio, fue el material utilizado en los anillos-O del transbordador el que perdió la capacidad de recuperar su forma original ante las bajas temperaturas de aquella mañana en Cabo Cañaveral y que, por lo tanto, no sellaron del modo especificado. El punto aquí es que, para Feynman, así como para cualquier otro científico o ingeniero, las especificaciones de un instrumento –o de un aparato tecnológico como es el transbordador– son una pieza de información fundamental sobre el diseño y uso apropiado de un instrumento, ya que contienen toda la información relevante sobre este.

Toda especificación, entonces, tiene un propósito *metodológico* y una función *epistémica*. Desde la perspectiva metodológica, funciona como un 'plan' para el diseño, construcción y uso de un instrumento. Desde la perspectiva epistémica, trabaja como un repositorio del conocimiento que los científicos e ingenieros responsables tienen sobre un instrumento, sus resultados e, incluso, sobre el origen de errores y fallas. En este contexto, una especificación tiene un objetivo doble: proveer la información relevante para la construcción y uso de un

instrumento y, al mismo tiempo, permitir comprender su funcionalidad.

En el contexto del software de computador, las especificaciones juegan un rol similar al que juegan en el del instrumento científico: son descripciones del comportamiento, los componentes y las capacidades del software de acuerdo con el sistema objeto.[2] Brian Cantwell Smith define las especificaciones como "una descripción formal en algún lenguaje formal, especificado en términos de un modelo, donde el comportamiento deseado es descrito" (Cantwell Smith, 1985, p. 20). El ejemplo utilizado es el de un sistema de reparto de leche, en el que debe ser especificado la manera en la que un camión de reparto visita cada tienda manejando la mínima distancia posible.

La definición de Cantwell Smith, aunque ilumina mucho la problemática, no captura bien lo que hoy los computólogos e ingenieros en sistemas llaman una 'especificación.' Hay al menos dos razones de que esto sea así. Por un lado, la definición es *reduccionista*. Por el otro, no es suficientemente inclusiva. Es reduccionista porque toda especificación se identifica con una *descripción formal* del comportamiento de un sistema objeto. Esto significa que el comportamiento esperado del sistema objeto es descrito en términos de un andamiaje formal. La ingeniería del software, por ejemplo, ha dejado en claro que las especificaciones no pueden ser completamente formalizadas. En cambio, descripciones 'semi-formales' del comportamiento del software deben ser incluidas, y eso por dos razones. Por un lado, porque los lenguajes formales no son suficientemente expresivos para describir la complejidad de muchos sistemas objetos. Por ejemplo, ningún lenguaje formal puede describir completamente una simulación computacional sobre el cambio climático. Por el otro lado, describir parcialmente el sistema objeto utilizando lenguaje natural u otra forma menos estricta de representación simplifica y naturaliza enormemente la especificación.

En este sentido, descripciones formales e informales del sistema objeto coexisten en la especificación. En otras palabras, fórmulas matemáticas y lógicas coexisten con instrucciones escritas en lenguaje natural, gráficos y hasta bosquejos del software. Juntamente con esto viene la documentación escrita en lenguaje natural –con algunos agregados formales– más los comentarios de los programadores. A pesar de que hay acuerdo generalizado en que una formalización total de la especificación es deseable, no es siempre posible, especialmente por el tipo complejo de especificación que se requiere para el software.[3]

Por el otro lado, la definición ofrecida por Cantwell Smith no es lo suficientemente inclusiva, puesto que especificar el comportamiento del sistema objeto no es independiente

del modo en que es implementado. Esto es, la especificación no describe totalmente el comportamiento esperado del software de acuerdo con el sistema objeto, sino que también incorpora restricciones teóricas y prácticas relacionadas con la computadora física. Esto significa, a su vez, que los científicos e ingenieros deben preocuparse también por incluir problemáticas asociadas a la computabilidad, la performance, la eficiencia y la robustez en la especificación. En el ejemplo de la simulación de reparto de leche, la especificación de Cantwell Smith solo captura aspectos generales del sistema objeto, ignorando cómo el sistema va a ser efectivamente implementado. Por ello es importante que se agregue a la especificación detalles sobre los límites y posibilidades de la computación del sistema. Como resultado, la especificación es poblada con nuevas entidades y relaciones, soluciones y alternativas a problemas que solo se resolverían de manera formal. Por ejemplo, si se espera que el camión recorra al menos 257 tiendas en su reparto de leche, entonces no puede ser implementado en una computadora de 8 bits y esto tiene que ser explicitado en la especificación.

Un ejemplo más complejo que se asocia con la anterior crítica de no poder formalizar completamente la especificación tiene que ver con el manejo de errores. Considérese la especificación de una simulación computacional de un satélite sujeto a estrés orbitando alrededor de un planeta, tal y como lo presentan Michael Woolfson y Geoffrey Pert en (1999). En esta simulación la subrutina Runge-Kutta tiene un error de discretización local del order de $\mathcal{O}(h^p + 1)$, y un error total acumulado del orden de $nCh^{p+1} = C(\bar{x} - x_0)h^p$. Ambas funciones de medición del error son iterativas, y por lo tanto saber el error local y total depende de cada iteración. De aquí que, aunque formalmente se pueda especificar la función de error, no se puede saber cuál será el error –y, por lo tanto, no puede ser formalizado el resultado de estas funciones de error (Durán, 2017, p. 39).[4]

Ahora bien, una idea más precisa de lo que constituye una especificación es ofrecida por Shari Pfleeger y Joanne Atlee (2009). Estas autoras creen que una especificación apunta a describir propiedades externas y visibles de la unidad de software, junto con las funciones, parámetros, valores de retorno y excepciones del sistema. En otras palabras, la especificación tiene en cuenta tanto el sistema objeto que se quiere implementar, así como los modos de implementación. Las autoras presentan una serie de atributos y características generales que son usualmente incluidas como parte de la especificación. Lo que sigue es solo una preselección de dichos atributos y características:

Propósito: se documenta la funcionalidad de cada función, variable, acceso I/O, etc. Esto debe ser realizado en detalle, a fin de que científicos e ingenieros puedan identificar rápidamente qué funcionalidad es más adecuada para sus necesidades.

Precondiciones: estas son un conjunto de supuestos que el modelo incluye y que deben estar disponibles a los científicos e ingenieros, a fin de saber bajo en qué condiciones el software funciona correctamente. Precondiciones incluyen valores de parámetros iniciales, estados de recursos globales y otras unidades de software;

Protocolos: estos incluyen información sobre el orden en el cual las funciones son invocadas, los mecanismos bajo los cuales los módulos intercambian mensajes, autorización de acceso restringido, etc.

Postcondiciones: los resultados de las funciones son documentados, incluyendo el valor de retorno de variables, excepciones, archivos de salida, etc. Postcondiciones son importantes porque indican cómo reacciona el sistema ante ciertos valores de entrada.

Atributos de calidad: estos son atributos de calidad de performance y confianza del modelo que deben estar visibles para desarrolladores y usuarios. En muchas simulaciones computacionales, por ejemplo, es necesario que el usuario especifique la variable de TOLERANCIA, es decir, el error máximo absoluto que es tolerado por el sistema. Si esta variable es instanciada con un valor muy bajo, el sistema puede tornarse muy lento (Woolfson and Pert, 1999);

Decisiones de diseño: la especificación es el en lugar donde son realizadas decisiones políticas, éticas y del diseño general del software son realizadas.

Tratamiento de error: es importante especificar el flujo de trabajo que tiene el software de computador y cómo se comporta en situaciones anormales, tales como valores de entrada inválidos, errores de cálculo, errores de manipulación, etc. Los requisitos de funcionamiento en la especificación deben indicar claramente lo que el software puede hacer en tales situaciones. Una buena especificación debe seguir principios específicos de robustez, corrección, completitud, estabilidad y desiderata similar.

De este modo las especificaciones de software computacional comparten funciones similares a los instrumentos científicos, ya que tienen un rol *metodológico* como planos para el diseño, codificación e implementación del software de computador. Incluido en este rol está el vincular la representación y el conocimiento sobre el sistema objeto junto con el conocimiento acerca del computador (i.e., la arquitectura del computador, el sistema operativo, los lenguajes de programación, etc.). Esto significa que las especificaciones ayudan a trazar una especie de puente conceptual entre estos dos tipos de conocimiento.

La especificación también tiene un rol *epistemológico* en el sentido de que es un repositorio del conocimiento que los ingenieros y científicos tienen sobre el sistema objeto y, como he mencionado, también sobre el computador también. En este sentido, la especificación es la unidad cognitiva más transparente del software.[5] Digo esto, porque los científicos e ingenieros siempre pueden comprender lo que ha sido descrito allí sobre el software. Para ver esto último, solo nos basta comparar la información contenida en la especificación con los algoritmos y con los procesos computacionales: con respecto a los primeros, a pesar de ser cognitivamente accesibles, están escritos en un lenguaje de programación que hace imposible para un individuo seguirlo completamente; respecto a los segundos, el acceso a los procesos computacionales es completamente imposible. Si bien es cierto que sin el acceso cognitivo que la especificación ofrece se pierde terreno para hacer sugerencias sobre la representación de los modelos computacionales, la corrección de los resultados y sobre nuestro conocimiento general obtenido a través del software de computador, es también cierto que desde la especificación a la visualización de los resultados hay varias instancias intermedias que contribuyen a oscurecer nuestro acceso general al software.[6]

Ilustremos estos puntos con un ejemplo. Considérese de nuevo la simulación computacional del satélite sujeto a estrés orbitando alrededor de un planeta que ofrecen Woolfson y Pert (Woolfson and Pert, 1999). Una especificación posible incluye información sobre el comportamiento del satélite, tales como el modo en el que se estira sobre el radio vector. También incluye los límites y las restricciones impuestas a la simulación. Woolfson y Pert indican que si la órbita no es circular, entonces el estrés del satélite varía y por lo tanto se expande y contrae a lo largo del radio vector de manera perfecta. Por estas razones, y a fin de que el satélite no se diseñe de manera perfectamente elástica, habrá efectos de histéresis y algo de la energía mecánica se convertirá en calor, que se pierde posteriormente. Una solución interesante para el problema de la elasticidad del satélite consiste en su rep-

resentación mediante tres masas, cada una con el mismo valor y conectadas por resortes de la misma longitud. Finalmente, una serie de ecuaciones que acompañen estos cambios deben ser incluidas en la especificación.

Hay también otros elementos que deben ser incluidos en la especificación junto con el conocimiento que tenemos sobre el computador. Un ejemplo que ilustra la importancia que tiene conocer la arquitectura sobre la cual se ejecuta el software es que, en el ejemplo de la simulación del satélite, la masa de Júpiter no puede ser representada en una arquitectura de 64 bits ya que la masa de este planeta es de $1.898x10^{27}$ kg y dicho número solo puede ser representado en, al menos, 128 bits.

Continuemos ahora con el estudio del algoritmo, la estructura lógica encargada de interpretar la especificación en un programa de computación adecuado.

3 Algoritmos

Mucha de nuestras actividades diarias pueden ser descritas como un conjunto simple de reglas que repetimos sistemáticamente. Nos despertamos a cierta hora del día, nos lavamos los dientes, nos duchamos y partimos para el trabajo. Por supuesto que modificamos nuestra rutina, pero solo lo suficiente como para que sea de algún modo más ventajosa de algún modo: nos dan más tiempo en la cama, minimiza la distancia entre paradas o satisface a todos en casa.

En cierto modo, estas rutinas diarias capturan lo que llamamos un *algoritmo* en el sentido que, para ambos casos (e.g., en la rutina y en el algoritmo) hay una repetición del mismo conjunto de acciones una y otra vez. Jean-Luc Chabert define un algoritmo como "el conjunto de instrucciones paso a paso a ser ejecutadas mecánicamente a fin de obtener un resultado deseado" (Chabert, 1994, p. 1). Así pues, la rutina antes descrita es, en cierto modo, un algoritmo.[7] Pero esto no es lo que científicos e ingenieros llaman un *algortimo*, asociado, más bien, con la práctica computacional. Por esto, en mi opinión, la noción de algoritmo descansa en la idea de que es parte de un procedimiento[8] sistemático, formal y finito.

Desafortunadamente, al caracterizar un algoritmo de la manera en que acabamos de hacerlo permite introducir estructuras que no consideraríamos, estrictamente hablando, como un algoritmo. Tomemos como ejemplo el siguiente sistema de partículas en celda sin colisiones:

1. Construya una grilla apropiada en un espacio de una, dos, o tres dimensiones, dentro del cual el sistema puede ser definido [...]

2. Decida el número de superpartículas, ya sean electrones o iones, y asigne a estas una posición. A fin de obtener una fluctuación mínima, se requiere que haya tantas partículas cuantas sean posible en cada celda [...]

3. Utilice la ecuación de Poisson para las densidades en ciertos puntos de la grilla: $\nabla^2\phi = -\rho/\varepsilon_0$

   ...

N. Si el tiempo total de la simulación no se ha excedido, retorne al punto *3*.[9]

Nótese que este ejemplo incluye declaraciones tanto en lenguaje natural como en lenguaje matemático. En algún sentido, se parece más a una especificación para un sistema de partículas en celda sin colisiones que a un algoritmo propiamente dicho. Sin embargo, bajo la definición ofrecida por Chabert, califica como un algoritmo. Ahora bien, como son de mi interés los algoritmos que se pueden implementar en la computadora, y claramente tal implementación no es posible con el algoritmo del ejemplo, entonces necesitamos precisar aun más la definición de algoritmo.

En la década del '30, el concepto de algoritmo se popularizó en el contexto de programación de una computadora. En este marco, la noción sufrió varias alteraciones, incluyendo el cambio del lenguaje base en el cual los algoritmos son escritos. En particular, una lista más o menos completa incluiría las siguientes características:

1. Un algoritmo es definido como un conjunto finito y organizado de instrucciones, que debe satisfacer cierto conjunto de condiciones con la intención de proveer soluciones a un problema;

2. Un algoritmo debe ser capaz de ser escrito en un determinado lenguaje;

3. El algoritmo es un procedimiento que es llevado a cabo paso a paso;

4. La acción de cada paso está estrictamente determinada por el algoritmo, la entrada de datos y los resultados obtenidos en pasos previos;

5. Cualesquiera que sean los datos de entrada, la ejecución del algoritmo debe terminar después de un número finito de pasos;

6. El comportamiento del algoritmo es físicamente instanciado durante la implementación en la computadora.[10]

El punto *2* es particularmente importante, puesto que menciona que el algoritmo debe ser escrito en un lenguaje específico. Aquí Chabert se está refiriendo a los distintos *lenguajes de programación* disponibles. Ahora bien, se sabe que el universo de lenguajes de programación es muy amplio e incluye muchos tipos distintos, dependiendo de las necesidades de los programadores, de lo que se pueda representar sobre el sistema objeto y de otras consideraciones como portabilidad, modularidad, sistemas operativos, etc. Aquí no nos importa realmente si el algoritmo se escribe en *pseudo-código*, en lenguaje *imperativo* (e.g., Fortran, C), en un lenguaje *interpretado* (e.g., Python), un lenguaje *funcional* (e.g., Haskell), o en otro lenguaje. Lo importante aquí es que los algoritmos poseen características distintivas que merecen nuestra atención.

Una de estas características interesantes de los algoritmos es que, desde un punto de vista ontológico, el algoritmo es una estructura sintáctica que codifica la información precisada en la especificación. Como discutimos más adelante, por razones técnicas y prácticas, un algoritmo no codifica toda la información contenida en la especificación. En cambio, algunas cosas son agregadas, otras perdidas, y otras simplemente alteradas. Más adelante volveré con mayor precisión sobre este punto que ahora es un tanto vago.

Desde el punto de visto ontológico, los algoritmos son *abstractos* y, en muchos casos, también *formales*. Son *abstractos* porque consisten en una secuencia de símbolos sin que actúen relaciones causales. Del mismo modo que una estructura lógico/matemática, los algoritmos son causalmente inertes y desconectados del nexus tiempo-espacio. Por otro lado, los algoritmos son formales en tanto que siguen las leyes lógico/matemáticas, las cuales establecen cómo manipular sistemáticamente la secuencia de símbolos que los componen. A pesar de que algunos autores están en desacuerdo con estas características de los algoritmos,[11] hay acuerdo generalizado en que son ontológicamente diferentes de las especificaciones y los procesos computacionales, que es lo que nos interesa establecer por

ahora. En la próxima sección, discuto con más detalle estas ideas y cuál es la preocupación de muchos filósofos con respecto a entender los algoritmos de manera diferente a como lo estamos describiendo aquí.

Estas características ontológicas de los algoritmos ofrecen ventajas epistémicas de mucho interés. Dos que han mantenido muy ocupados a los filósofos e informáticos son la *correlación sintáctica* y la *transferencia sintáctica*. La correlación sintáctica consiste en la posibilidad de mantener equivalencia entre dos estructuras algorítmicas. La transferencia sintáctica, en cambio, consiste en alterar el algoritmo de tal manera que realice una funcionalidad diferente. Discutamos ahora con más detalle estos dos puntos.

Comencemos con un ejemplo de correlación sintáctica que va a resultar familiar. Considérese un sistema cartesiano de coordenadas y un sistema polar de coordenadas. Sabemos que hay transformaciones matemáticas bien estudiadas que permiten establecer una equivalencia entre los dos sistemas de coordenadas. Dado un sistema de coordenadas (x, y), su equivalente en coordinadas polares es obtenido por $(r, \theta) = (\sqrt{x^2 + y^2}, \tan^{-1} \frac{y}{x})$. Del mismo modo, dado un conjunto de coordenadas polares (r, θ), podemos encontrar su correspondiente en coordenadas cartesianas sin mucho esfuerzo del siguiente modo $(x, y) = r\,cos\theta, r\,sin\theta)$.[12]

La misma idea es extensible al caso de los algoritmos. Considérese ahora el condicional en el algortimo 1 y su equivalente en el algoritmo 2. Sabemos que ambos son lógicamente isomórficos y, por lo tanto, equivalentes ya que hay una prueba formal que así lo establece (véase la tabla de equivalencias 1).[13]

Algorithm 1
. . . **if** (A) **then** $\{a\}$ **else** $\{b\}$ . . .

Algorithm 2
. . . **if** (**not**-A) **then** $\{b\}$ **else** $\{a\}$ . . .

Preguntémonos ahora, ¿cuáles son las ventajas epistémicas de la correlación sintáctica? Una ventaja es que expande el número de posibles –y equivalentes– implementaciones de un software. Considere el caso de un conjunto de ecuaciones Langraniano y uno Hamiltoniano. Los criterios para elegir uno sobre el otro para representar sistemas dinámicos puede ahora depender de necesidades pragmáticas y no epistémicas. Es decir, depende de la performance

Table 1: Tablas de verdad para la equivalencia de los algoritmos 1 y 2.

A	$\{a\}$	$\{b\}$
T	T	$\emptyset$
F	$\emptyset$	T

not-A	$\{a\}$	$\{b\}$
F	T	$\emptyset$
T	$\emptyset$	T

del software o de la facilidad con la que se entiende uno respecto del otro, y no de su capacidad de representación. En este sentido, los científicos no están obligados a hacer uso de un sistema de ecuaciones cuando se puede demonstrar que son equivalentes a otro sistema de ecuaciones que es, en su conjunto, más simple de interpretar, más eficientes, etc. Para ilustrar este punto, hagamos uso del siguiente cálculo. Para un sistema de n dimensiones, las ecuaciones Hamiltonianas son un conjunto de $2n$ Ecuaciones Ordinarias Diferenciales (EOD) de primer orden acopladas, mientras que las Lagranianas son un conjunto de n EOD de segundo orden desacopladas. Así, un algoritmo que use ecuaciones Hamiltonianas en lugar de Lagranianas será más eficiente en términos de performance, uso de memoria, y velocidad en los cálculos. Algo muy similar ocurre con el ejemplo de los dos sistemas de coordenadas mencionados anteriormente, en donde el sistema cartesiano es más fácil de comprender y calcular por un individuo que el sistema polar.

La segunda característica epistémica de los algoritmos es la *transferencia sintáctica*. Esta se refiere a la idea de que mediante la adición –o sustracción– de algunas líneas de código en el algoritmo es posible reutilizar el mismo algoritmo en diferentes contextos. Un ejemplo simple consiste en expandir un algoritmo que suma dos números a un algoritmo que, con unas pocas líneas de código extra, suma n números. Transferencia sintáctica permite reutilizar código existente a fin de acomodarlo a distintos contextos y, posiblemente, generalizarlo. Nótese que transferencia sintática es la idea fundamental detrás de la modularización del software mediante librerías, I/O, rutinas, etc.

Correlación y transferencia sintáctica son prácticas comunes entre científicos e ingenieros que programan su propio software. En este sentido, es frecuente ver que el software crece y se reduce a medida que se le agregan algunos módulos y se modifican otros. Esto es, en verdad, parte estándar del mantenimiento de código.

Como indiqué al principio, ambas formas de modificar los algoritmos vienen con problemas filosóficos incluidos. El más evidente es la pregunta "¿cuándo dos algoritmos son el mismo?". Esta pregunta surge en el contexto de correlación y transferencia sintáctica,

puesto que se presupone, como ya he dicho, modificaciones en el algoritmo original que llevan a un nuevo algoritmo. En el caso de correlación sintáctica, esto viene de la mano de nuevos algoritmos que llevan adelante las mismas funcionalidades que el viejo algoritmo. En el caso de transferencia sintáctica, esto resulta de la modificación de un algoritmo y, por lo tanto, del cambio de su identidad.

Hay dos posibles respuestas a la pregunta anterior. O bien los dos algoritmos son *lógicamente equivalentes*, en cuyo caso los dos algoritmos son estructuralmente similares,[14] o bien son *conductualmente equivalentes*, esto es, los algoritmos se comportan en ciertas condiciones de la misma manera. Discutamos ahora brevemente cada una de estas respuestas.[15]

La equivalencia lógica indica que dos algoritmos son estructuralmente similares y que su equivalencia puede ser mostrada de manera formal. Ya ilustramos un caso simple de equivalencia lógica usando los algoritmos 1 y 2, pues ambos son estructuralmente isomórficos entre sí. Procedimientos formales de cualquier tipo –como la tabla de verdad 1– son garantías epistémicas de equivalencia lógica.[16] Así, el condicional $if \ldots then$ del algoritmo 1 es estructuralmente equivalente al condicional en el algoritmo 2, y la tabla de verdad 1 muestra su equivalencia lógica.

Desafortunadamente, equivalencia lógica no es siempre posible a causa de restricciones prácticas y teóricas. Ejemplos de restricciones prácticas incluyen casos en donde la equivalencia lógica de dos o más algoritmos no puede ser establecida o verificada formalmente, porque, por ejemplo, dichos procedimientos formales consumen demasiado tiempo y recursos. Ejemplos de restricciones teóricas, por el otro lado, incluyen casos en los que el lenguaje de programación se refiere a entidades, relaciones, y operaciones sobre los cuales es imposible de verificar equivalencia lógica con otro algoritmo, por ejemplo, porque el lenguaje formal, supongamos, no es lo suficientemente expresivo (véase, por ejemplo (Turner and Angius, 2017)).

A fin de solucionar estas restricciones, la academia y la industria han unido fuerzas y han creado una familia de herramientas que automatizan el procedimiento de equivalencia lógica mediante pruebas y chequeo. Un buen ejemplo de estos procedimientos utilizado en la verificación de algoritmos es ACL2 (A Computational Logic for Applicative Common Lisp), diseñado para llevar adelante razonamientos automatizados y, por lo tanto, para ayudando en la reconstrucción de clases de equivalencia en algoritmos.

Por el otro lado tenemos equivalencia conductual, la cual consiste en asegurar que los

dos algoritmos se comporten de manera similar para alguna definición de 'similar' (e.g., producen los mismos resultados). A pesar de que la equivalencia conductual parece ser más simple de lograr que la equivalencia lógica, principalmente porque esta última requiere estructuras y procedimientos formales que sustenten la equivalencia, aun así presenta varios problemas filosóficos y técnicos de interés. Por ejemplo, la equivalencia conductual está basada en principios inductivos, con lo cual uno solo podría garantizar equivalencia hasta tiempo t (i.e., cuando los dos algoritmos se comportan de manera similar), pero no hay garantías de que en el siguiente paso $t + 1$ el comportamiento de ambos algoritmos sea el mismo.

En este sentido, la equivalencia conductual solo puede ser garantizada hasta el momento en que los dos algoritmos se comparan. De hecho, y si nos ponemos estrictos, la equivalencia conductual solo puede ser garantizada para *una* ejecución específica del algoritmo. Otras ejecuciones del mismo algoritmo podrían mostrar un comportamiento diferente y, por lo tanto, socavar los ideales de equivalencia. Es cierto que muchos filósofos son renuentes a aceptar como problema genuino el hecho de que cada algoritmo se comporta de una manera diferente en cada ejecución. Sin embargo, la pregunta está planteada y debe ser respondida. Por esto, retomo estas discusiones en el contexto de verificación de programas, que es a donde pertenecen (véase la sección 4).

Si recapitulamos, notamos por último que la equivalencia conductual puede ocultar alguna forma de equivalencia lógica. Esto significa que dos algoritmos divergen en comportamiento a pesar de ser lógicamente equivalentes. Un ejemplo es un algoritmo que implementa coordenadas cartesianas y otro que implementa coordenadas polares. Ambos algoritmos son estructuralmente equivalentes y, dada su simpleza, podríamos especular que es posible mostrar los procesos formales que permiten la equivalencia lógica, pero sin embargo se comportan de manera diferente.[17] En este tipo de casos, la pregunta es qué tipo de equivalencia debe prevalecer.

Estas son algunas discusiones que están en el corazón de la llamada filosofía de la ciencia de la computación (Turner and Angius, 2017). Claro que aquí solo he presentado algunos aspectos de la cuestión y, debo admitir, de manera un tanto general. Sin embargo, nuestra tarea no era discutir en profundidad cada uno de estos temas sino más bien mostrar cómo la filosofía permite comprender problemas que muchas veces son erróneamente abordados como netamente técnicos (e.g., computacionales, técnicos, matemáticos, o lógicos, pero no filosóficos).

Lo que nos queda ahora por discutir es el vínculo que hay entre la especificación y el algoritmo. Idealmente, la especificación y el algoritmo deben estar fuertemente relacionados, esto es, la especificación debe estar completamente interpretada en la estructura de un algoritmo. En la realidad, sin embargo, esto raramente sucede ya que la especificación hace uso de lenguaje natural que hace difícil la interpretación literal en un algoritmo. Un ejemplo es el uso de metáforas y analogías. Muchos modelos científicos e ingenieriles incluyen términos que no tienen una interpretación específica, como 'agujero negro' o 'mecanismo.' Metáforas y analogías son utilizadas para llenar un hueco conceptual en el modelo o teoría y así pasan a la especificación. En este sentido, las metáforas y analogías inspiran un proceso creativo en los usuarios de las especificaciones, pero que no pueden ser reconstruidos en el lenguaje literal utilizado en el algoritmo, ya que, de lo contrario, no es posible el cómputo.

Pero aún hay más. Anteriormente mencioné que la especificación incluye constructos no-formales, tales como el conocimiento y la experiencia del experto, las decisiones de diseño (e.g., políticas, morales, conceptuales, etc.) que no pueden ser formalmente interpretadas. A pesar de esto, dichos constructos no-formales necesitan ser incluidos correctamente en el algoritmo, caso contrario no serán parte de la computación del software. Imagínese, por ejemplo, la especificación de una simulación de sistemas de votos. Para que esta simulación sea exitosa, se implementan módulos estadísticos, de modo que den una distribución razonable de la población que vota. Durante las etapas de especificación, continúa nuestro ejemplo, los investigadores deciden dar más relevancia estadística a variables tales como sexo, género y acceso a sistemas de salud sobre otras variables, tales como educación y salarios. Si las decisiones de diseño no son programadas correctamente en el módulo de estadísticas, entonces la simulación nunca reflejará estas variables a pesar de estar en la especificación.

Este ejemplo apunta a mostrar que un algoritmo debe ser capaz de interpretar constructos formales, así como no-formales incluidos en la especificación. Esto es particularmente importante porque no hay métodos formales para interpretar el conocimiento y experiencia del experto, sus experiencias pasadas y presentimientos sobre un sistema computacional.

Desde luego, y para hacer más simple la interpretación de la especificación en el algoritmo, es usual hacer uso de lenguajes especializados que formalizan la especificación facilitando de este modo su interpretación como un algoritmo. Common Algebraic Specification Language (CASL), Vienna Development Method (VDM), Abstract Behavioral

Specification (ABS), y la notación Z son solo algunos ejemplos.[18] Model checking es útil ya que testea automáticamente que el algoritmo cumpla con las especificaciones indicadas y, por lo tanto, facilita su interpretación. Así pues, la interpretación de la especificación en un algoritmo tiene una larga tradición en matemáticas, lógica y ciencia de la computación, y no representa un problema conceptual ni práctico (i.e., existen y se conocen los métodos para una interpretación exitosa de la especificación en el algoritmo), aunque sí trae consigo algunos problemas filosóficos (e.g., asegurarse que la formalización de la especificación sea representativa del sistema objeto).

¿Dónde nos deja esta discusión en el mapa metodológico y epistemológico del software de computador? Por un lado, tenemos una mejor idea de la naturaleza de las especificaciones y los algoritmos como unidades de análisis y sus procesos creativos y de diseño implicados en la práctica científica e ingenieril. Por el otro lado, tenemos una idea más acabada de cuál es el alcance y los límites epistémicos de la especificación y los algoritmos, así como una comprensión más específica del origen de posibles problemas que afecten la credibilidad de los resultados de una computación.

4 Procesos computacionales

En la primera sección, discutimos brevemente la definición de *especificación* ofrecida por Cantwell Smith y de las dificultades que esta presentaba. Aquel análisis nos permitió entender la importancia de las especificaciones en el software de computador y de cómo se situaban en el mapa metodológico y epistemológico. Ahora quiero complementar aquella discusión con un análisis de los *programas computacionales* ofrecido por el mismo autor. El formato de esta sección es similar al presentado durante la discusión de especificaciones, es decir, primero discuto la definición de Cantwell Smith, para luego mostrar sus dificultades y cómo superarlas.[19]

Parafraseando a Cantwell Smith, un *programa computacional* es un conjunto de instrucciones que tienen lugar en una computadora física. Notemos que esta caracterización se diferencia bastante de su noción de especificación. Mientras que un programa computacional es un proceso causalmente relacionado que tiene lugar en una computadora física, la especificación es una unidad de análisis más bien abstracta y, para algunos, incluso formal. Además, Cantwell Smith indica que "el programa tiene que decir *cómo el comportamiento [de la especificación] se obtiene*, típicamente de un modo step-by-step (y usualmente en

mucho detalle). La especificación, por el contrario, es menos restringida: todo lo que tiene que hacer es especificar *cuál es el comportamiento apropiado*, independientemente de cómo se logra" (Cantwell Smith, 1985, p. 22. Énfasis en el original.). Así, las especificaciones son *declarativas* en el sentido que explicitan claramente en un lenguaje dado –natural y formal– cómo el sistema se desarrolla en el tiempo. A este respecto, las especificaciones denotan lenguajes de alto nivel para aproximarse a problemas sin requerir explicitar la naturaleza exacta del procedimiento a seguir. Programas computacionales, por el otro lado, son *procesales* en el sentido de que determinan paso a paso el modo en que la computadora física se debe comportar.

A fin de ilustrar las diferencias entre la especificación y el programa computacional, tomemos de nuevo la descripción de una simulación de un camión de reparto de leche. En esta simulación, el camión debe distribuir leche a cada tienda manejando la distancia más corta posible. De acuerdo con Cantwell Smith, esta es una descripción de lo que debe suceder, pero no de *cómo* sucede. Es el programa computacional el responsable de mostrar cómo el camión, de hecho, reparte la leche: "maneje cuatro cuadras al norte, doble a la derecha, párese en la tienda de Gregory en la esquina, deje la leche, luego maneje diecisiete cuadras al noreste, [...]" (Cantwell Smith, 1985, p. 22).

A pesar de que esta caracterización es correcta en muchos sentidos, la definición de Cantwell Smith de 'programa computacional' no es lo suficientemente persuasiva. El problema principal es que no capta la diferencia entre un proceso paso a paso entendido como una fórmula sintáctica (i.e., el algoritmo) de un proceso paso a paso que pone la máquina física en los estados causales apropiados (i.e., el proceso computacional).[20] En particular, no reconocer esta distinción está en la base de confundir la descripción de un sistema de reparto de leche –en forma de un algoritmo– y los estados físicos en los que entra una computadora cuando ejecuta la descripción de un sistema de reparto de leche (en forma de un proceso computacional). Notemos que esta no es una diferencia ociosa, sino que apunta al problema principal de la verificación de software, como veremos más adelante.

La noción de programa computacional, entonces, necesita ser dividida a su vez en dos: el algortimo y el proceso computacional. Dado que ya hemos discutido la noción de algoritmo con cierto detenimiento, ahora vamos a discutir la noción de proceso computacional y su relación con los algoritmos. Comencemos con la relación entre procesos computacionales y algoritmos, ya que nos permite entender mejor la noción de proceso computacional en sí misma.

Como muchos científicos e ingenieros que programan saben muy bien, implementar un algoritmo en una computadora implica compilarlo primero. Compilar básicamente consiste en un mapeo de un dominio sintáctico (i.e., el algoritmo) en un dominio semántico (i.e., el proceso computacional)[21]. En otras palabras, un algoritmo es implementado como un proceso porque la computadora puede interpretarlo y ejecutarlo de la manera correcta. ¿Cómo, entonces, es esto posible?

En su funcionamiento básico cada hardware de computador consiste en mircoelectrónica construida a partir de millones de puertas lógicas. Estas puertas lógicas son la implementación lógica de los operadores 'and', 'or' y 'not' que, cuando se combinan, permiten llevar adelante operaciones lógicas y aritméticas y, por lo tanto, todas las demás operaciones que se ejecutan. En este nivel de descripción, todas las computadoras son básicamente las mismas, con la excepción del número de puertas lógicas utilizadas. Sin embargo, hasta aquí llegan las comparaciones, ya que, a medida que se complejiza la descripción de la arquitectura, las computadoras como artefactos adquieren mayor complejidad y por lo tanto difieren entre sí.

Un planteamiento de abajo hacia arriba conecta estas puertas lógicas con los procesos computacionales, incluyendo una cascada de instrucciones de computador muy complejas y de lenguajes que 'hablan' entre sí. Una descripción breve de cómo sucede esto sería: Microcode es un lenguaje que permite implementar a nivel muy bajos de hardware las instrucciones de más alto nivel (e.g., lenguaje Assembly); el compilador, que es responsable de convertir las instrucciones del algoritmo en código de máquina, de manera que puedan ser leídas y ejecutadas como un proceso computacional por la computadora. Así, un proceso computacional se ejecuta en una computadora física, en donde hay varias capas de intérpretes que traducen el conjunto de instrucciones del algoritmo en un lenguaje de hardware apropiado.

Ilustremos estos puntos con un caso simplificado de la operación matemática $2 + 2$ escrita en lenguaje C. Considérese el siguiente algoritmo 3.

Algorithm 3 Un algoritmo simple para la operación $2 + 2$ escrita en lenguaje C

```
void main()
{
  return(2+2)
}
```

En código binario, el número 2 es representado por 00000010, mientras que la adición puede ser llevada a cabo por un Ripple-carry adder, por ejemplo. El compilador convierte las instrucciones del algoritmo 3 en código de máquina listo para ser ejecutado en la computadora física. Una vez que el proceso computacional finaliza, se mostrará en la pantalla el resultado, que en este caso es 4 o, en código binario, 00000100.

Tomando estas ideas, tenemos suficientes elementos para discutir algunos temas filosóficos de importancia. Ya hemos discutido la ontología de los algoritmos y lo que esto significa para una evaluación epistemológica. Aquí vamos a hacernos preguntas similares. Considérese el siguiente problema: ¿cómo es posible dar sentido a la idea de que un proceso computacional –que es un proceso físico en la computadora– implementa un algoritmo que es una representación abstracta? Desde un punto de vista epistemológico podemos decir que un proceso computacional y un algoritmo son *epistémicamente* equivalentes. Pero esto, a su vez, plantea el siguiente interrogante: ¿en qué sentido específico podemos decir que los procesos computacionales y los algoritmos son epistémicamente equivalentes? Hay una respuesta simple e intuitiva, y es que son equivalentes en el sentido de que la información codificada en el algoritmo es físicamente instanciada por la computadora. El ejemplo de la adición de $2 + 2$ es codificada en el algoritmo 3. Asumiendo la correcta funcionalidad del compilador y de la computadora física, el resultado de la computación del algoritmo es la adición 4.

De aceptar la equivalencia epistémica entre algoritmos y procesos computacionales surgen algunas preguntas de corte filosófico. Una de particular interés es la relacionada con la verificación de los algoritmos. La cuestión aquí es preguntarse si un algoritmo verificado implica un proceso computacional también verificado. Responder a esta pregunta afirmativamente significa que hay un modo formal de garantizar que el proceso computacional se comporte del modo especificado y, por lo tanto, que los resultados sean correctos con respecto a dicha especificación. Es más, ello implicaría que las tres unidades de análisis del software computacional constituyen una única unidad epistémica. Ahora bien, responder negativamente pone mucha presión en el modo en que la información codificada en el algoritmo es físicamente instanciada. En otras palabras, surge una pregunta respecto de los resultados del software computacional, si reflejan o no aquello que se ha codificado en el algoritmo en primer término. Desde luego, aunque los métodos de validación sean pilares fundamentales para la confianza de los resultados del software, no son los únicos (i.e., métodos de validación también juegan un rol fundamental en la confianza que se gen-

era sobre los resultados).[22] Discutamos ahora el debate sobre verificación formal en más detalle.

Los informáticos y filósofos C. A. R. Hoare (1999) y E. J. Dijkstra (1974) consideran que el software de computador tiene una naturaleza matemática. En este contexto, algoritmos –así como procesos computacionales– pueden ser formalmente verificados, esto es, se puede mostrar que el algoritmo es correcto con respecto a ciertas propiedades formales de la especificación, algo así como una prueba formal. Una diferencia importante con las matemáticas, sin embargo, es que la verificación de software de computador requiere de su propia sintaxis, la cual difiere en muchos sentidos de la utilizada en matemáticas. Para lograr esto, Hoare creó sus famosas ternas, que consisten en un sistema formal –estados iniciales, intermedios, y finales– que obedecen, estrictamente, un conjunto de reglas lógicas. Las ternas de Hoare tienen la siguiente forma: $\{P\}\ C\ \{Q\}$ donde P y Q son predicados lógicos –*precondiciones* y *postcondiciones*– y C es un programa. Cuando las precondiciones se cumplen, el programa C se ejecuta y se establece la poscondición. Los predicados son formulas en lógica de predicados con reglas específicas. Una de estas reglas es el predicado vacío: $\frac{}{\{P\}Skip\{P\}}$; otro es la asignación de valores a una variable: $\frac{}{\{P[E/x]\}x{:=}E\{P\}}$, y así Hoare (1971).

Para citar a Hoare y Dijkstra, los algoritmos y los procesos computacionales son "confiables y obedientes" (Dijkstra, 1974, p. 608), ya que se comportan exactamente como se instruyó en la especificación. En este sentido, científicos e ingenieros solo necesitan preocuparse por verificar el algoritmo, pues el proceso computacional que lo implementa está garantizado como confiable.

El punto de vista contrario sostiene que los algoritmos, al ser ontológicamente diferentes de los procesos computacionales, también difieren epistémicamente. Este argumento sostiene que cuando un algoritmo, cuando es implementado como un proceso computacional, se convierte en un proceso causal –perdiendo, de algún modo, su naturaleza abstracta y formal– y, por lo tanto, sujeto a las dificultades usuales de procesos físicos. Por estas razones, un algoritmo verificado no implica un proceso computacional verificado. La conclusión es que los algoritmos y los procesos computacionales no puede ser epistemológicamente similares. Mientras que los primeros son formulaciones de naturaleza matemática –o lógica– apropiados para verificación, la corrección de los procesos computacionales solo puede ser abordada usando métodos empíricos. Estas ideas están en la base del argumento utilizado por James Fetzer (1988), quien considera que los procesos com-

putacionales son causales en su naturaleza y, por lo tanto, caen por fuera del ámbito de métodos lógicos y matemáticos.[23] Esto significa que, cuando el algoritmo es implementado en la máquina física donde surgen los procesos causales, el software computacional se torna 'causal.' El argumento termina con la idea de que la verificación formal es imposible en informática y que a los procesos de validación como testing se les debe dar mayor importancia que la que actualmente tienen.[24]

Muchos informáticos y filósofos objetaron vehementemente los argumentos de Fetzer, acusándolo de no entender las bases de la teoría de computación.[25] Es más, si la idea de Fetzer es correcta, muchos argumentaron, entonces lo mismo se debería decir sobre las matemáticas puesto que estas deben realizarse en algún medio físico (e.g., nuestro cerebro, una calculadora, un ábaco). El argumento de Fetzer, entonces, solo funciona si aceptamos una diferencia cualitativa entre el medio físico en el que se implementa un algoritmo (i.e., la computadora física) y un medio para llevar a cabo una prueba matemática (e.g., nuestro cerebro) (Blanco and García, 2011).

A pesar de estas objeciones, creo que Fetzer tiene la razón, al menos, en dos puntos. Primero, está en lo correcto al señalar que los algoritmos y los procesos computacionales no pueden ser conceptualizados como una misma entidad matemática, sino que requieren un tratamiento distinto. Ya mencioné esto anteriormente, cuando separé ontológicamente los algoritmos como entidades abstractas –y formales– de procesos computacionales causalmente relacionados. Sin embargo, creo que está errado en que estas diferencias constituyen razones para rechazar verificación formal y equivalencia epistémica entre algoritmos y procesos computacionales. Hoy en día, muchos algoritmos son y necesitan ser verificados. Por ejemplo, los protocolos en cryptografía utilizados en bancos y sistemas de seguridad. Lo mismo sucede con el software que controla aviones, satélites y transbordadores espaciales.

Segundo, Fetzer destaca el rol de validación como más relevante de lo que se pensó originalmente. Yo estoy de acuerdo con él en este punto. Métodos de validación no pueden ser absorbidos y reemplazados por verificación formal, aun cuando estos sean posibles. De hecho, y en el contexto de software de computación, una combinación de métodos de verificación y validación son fuertemente utilizados para establecer la confianza que se tiene en el software.

La figura 1 resume las tres unidades del software de computador discutidas hasta aquí, así como las conexiones entre ellas. Primero encontramos la especificación, en donde son

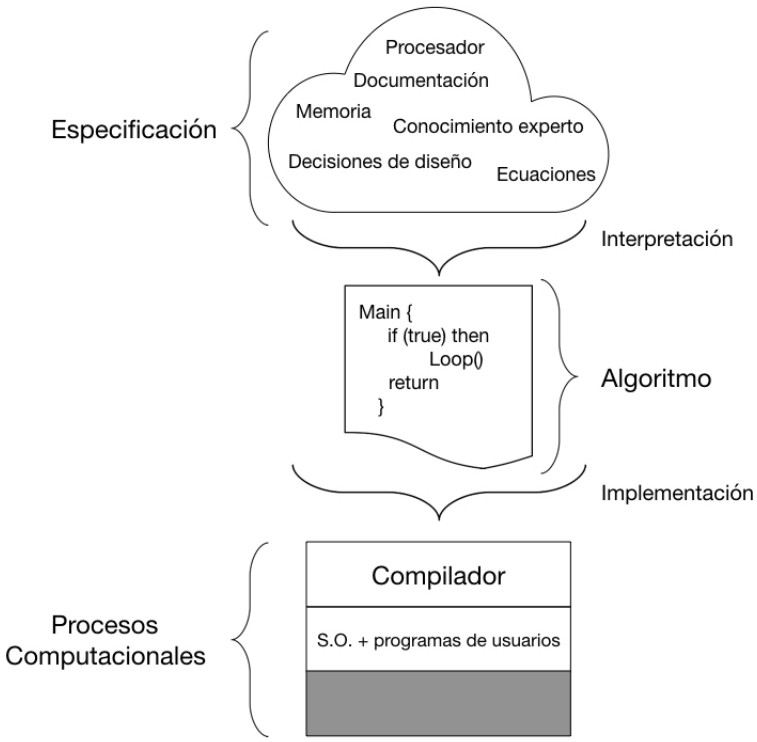

Figure 1: Un esquema general de la estructura del software de computador.

llevadas a cabo las decisiones sobre cómo será el software. Luego le sigue el algoritmo, en el que encontramos el conjunto de instrucciones que interpretan la especificación y prescribe cómo la computadora ha de comportarse, ilustrado en el proceso computacional al final de la figura.

5 Conclusión

La discusión que he presentado sirve como un mapa general para localizar el software de computador y sus tres unidades de análisis. Además, he identificado varios problemas filosóficos relacionados con cada unidad de análisis, así como la relación que tienen entre sí. Entre los más relevantes, discutí la definición de especificación y de programa

computacional de Cantwell Smith, así como la necesidad de distinguir entre algoritmos y procesos computacionales. Asimismo, discutí tanto la metodología y epistemología de especificaciones como el rol que juegan en nuestro entendimiento general del software de computador. En cuanto a la noción de algoritmo, centré mis fuerzas en entender su naturaleza y su relación con entidades matemáticas, un tema aun en discusión en estudios ontológicos y epistemológicos del software. Como vimos, de especial importancia fue el problema de equivalencia de dos algoritmos. También presenté y discutí el debate sobre verificación formal y justificación de procesos computacionales. La discusión aquí se focalizó en problemas de justificación y ontología de procesos de computación.

Por supuesto, todavía estamos lejos de haber abordado todos los problemas filosóficos que el software de computador sugiere, y aún más lejos de profundizar cada uno de ellos. Mucho queda por hacer, pero confío en que este trabajo motivará futuras discusiones.

Notes

[1]Otros análisis metodológicos y epistemológicos del software incluyen más niveles de análisis. Giuseppe Primiero entiende que se pueden identificar hasta seis de estos niveles de análisis, a saber: intencionalidad, especificación, algoritmo, instrucciones de lenguaje de programación de alto orden, operaciones de código de máquina/Assembly, y ejecución Primiero (2016). Aquí sintetizo la intencionalidad como parte de la especificación, las instrucciones de lenguaje de programación de alto orden como parte del algoritmo, y las operaciones de código de máquina/Assembly y su ejecución como parte de los procesos de computación.

[2]Aquí estoy dejando fuera de consideración la computadora física, la cual también requiere de una especificación.

[3]Queda sin responder la pregunta de si es posible reducir el lenguaje no-formal en lenguaje formal. La idea es que un lenguaje no-formal solo sirve para 'presentar' lo que es "formalizable en principio." Como aquí tomo una postura más cercana a la ingeniería del software, mi respuesta puede anticiparse como negativa. Sin embargo, la postura opuesta ha sido y continúa siendo defendida por muchos (por ejemplo, (Dijkstra, 1974; Hoare, 1996)). Agradezco a un/a evaluador/a anónimo/a por sugerir esta pregunta.

[4]En (Durán, 2017) hay también una especificación del algoritmo proporcionado por Woolfson y Pert.

[5]Por supuesto esto no significa que una especificación completa es siempre posible. De hecho, se podría argumentar que este es rara vez el caso. Dado que estos temas tienen que ver más con estudios sobre la práctica y la sociología de la ciencia que con lo que estamos tratando aquí, no voy

a discutir este punto con detenimiento.

[6]Estudios filosóficos más detallados sobre la naturaleza de la especificación pueden encontrarse en Raymond Turner (2011) y William Rapaport (1999). Aquí solo estoy interesado en identificar el rol metodológico y epistemológico de la especificación.

[7]Estrictamente hablando, una rutina no es un algoritmo. La analogía se sostiene, si se toma la noción de algoritmo como algo muy general. En cualquier caso, esto no afecta el argumento de esta sección.

[8]Este procedimiento puede ser matemático, lógico o de alguna otra naturaleza, dependiendo de nuestros compromisos ontológicos con la noción de algoritmo. Por ejemplo, C. A. R. Hoare (1989) suscribe a los algoritmos como entidades matemáticas, mientras que James Fetzer, por el contrario, los concibe como estructuras lógicas (Fetzer, 1988).

[9]Véase (Woolfson and Pert, 1999, 115) para una discusión detallada de los pasos involucrados en este proceso.

[10]Estas son algunas de las características que Chambert le adscribe a los algoritmos. Para más detalles, véase (Chabert, 1994, p. 455).

[11]Ammon Eden sugiere que hay tres paradigmas de la ciencia de la computación estructurados dependiendo de sus metodologías, epistemología y ontología. Es en este último concepto donde se inscriben las distintas formas de entender los algoritmos. Siguiendo a Eden, el paradigma *racionalista* entiende los algoritmos como expresiones matemáticas (Eden, 2007, 145); el paradigma *tecnocrático* los entiende como "montones de datos" y en ese sentido no muy distintos del *Hamlet* de Shakespeare (Eden, 2007, 153); finalmente, el paradigma *científico* entiende los algoritmos como alguna forma de objeto científico tales como entidades mentales, DNA, etc. En este sentido, los algoritmos tienen un conjunto único de propiedades que los caracterizan, tales como temporalidad, no fisicalidad, causalidad, etc. (Eden, 2007, 159).

[12]Notemos que este ejemplo motiva la siguiente pregunta: '¿hasta qué punto estos dos sistemas son equivalentes?' En este caso, se podría objetar que ciertas restricciones deben ser impuestas al sistema de coordenadas (i.e., para la función *tangente*, el domino son todos los números reales excepto $\pm\frac{\pi}{2}, \pm\frac{3\pi}{2}, \pm\frac{5\pi}{2}, ...$, donde la función no está definida. En este sentido el sistema cartesiano y el polar no son completamente isomórficos, sino solo parcialmente.

[13]Desde luego, los algoritmos son estructuras mucho más complejas que los ejemplos aquí presentados. No es una sorpresa, por lo tanto, que la equivalencia entre algoritmos no pueda ser establecida mediante tablas de verdad. Un modo de solventar este problema consiste en recurrir a herramientas ofrecidas en el área de matemáticas y teoría de la computación. Por ejemplo, un modo de lograr esto es construyendo una clase de equivalencia de algoritmos (Blass et al., 2009). El ejemplo es un algoritmo de ordenamiento (e.g., Bubble sort), el cual retorna una permutación ordenada de una lista, para alguna definición de orden. Mostrando que una función dada tiene la propiedad de devolver una permutación ordenada –para la misma definición de orden–, entonces estamos en

condiciones de decir que ambos algoritmos pertenecen a la misma clase. Encontrar equivalencia de algoritmos es central para muchos procedimientos de verificación de software y hardware.

[14]Aquí isomorfismo se presenta como la mejor opción ya que es el único –morfismo que puede garantizar equivalencia total entre algoritmos. Otras formas de –morfismo son también discutidas en la bibliografía especializada, como por ejemplo en el trabajo de Blass et al. (2009) y Blass and Gurevich (2003).

[15]Los trabajos de Robin Hill (2015) y Raymond Turner (2014) exponen lo complejo que puede ser la noción de algoritmo y, por lo tanto, cuándo dos algoritmos son el mismo.

[16]Hay varios problemas filosóficos relacionados con la idea de 'similaridad estructural' que no podemos abordar aquí. Sobre estos temas, véase por ejemplo (Humphreys and Imbert, 2012).

[17]Notemos que el argumento depende de qué entendamos por 'comportamiento'. Si aquí entendemos que dos algoritmos se comportan de manera similar cuando producen los mismos resultados, entonces claramente el algoritmo que implementa coordenadas polares se comporta diferente del algoritmo que genera coordenadas cartesianas. Desde luego, otras formas de definir 'comportamiento' son posibles y hasta más interesantes.

[18]Sobre CASL, véase Bidoit and Mosses (2004). Sobre VDM, véase Bjorner and Henson (2007). Sobre ABS, véase `http://abs-models.org/concept/`. Sobre la notación Z, se recomienda el clásico Spivey (2001).

[19]Es importante señalar que aquí no me preocupo por la arquitectura del computador en donde se ejecuta el algoritmo, y por lo tanto, el proceso computacional. Debería quedar claro a partir del contexto, de todas maneras, que aquí me refiero a computadoras basadas en circuitos integrados en oposición a computadoras cuánticas o biológicas.

[20]A fin de completar las fases que van desde la especificación al programa computacional, debemos agregar una serie de procesos intermedios, tales como compilar el algoritmo, asignar memoria, etc. Dado que estos procesos no constituyen, estrictamente hablando, unidades de análisis del software, no las vamos a considerar aquí. Desde luego, no por ello son menos importantes.

[21]Por detalles en estos dos puntos, véase (Rapaport, 1999).

[22]Para una discusión más profunda sobre justificar los resultados de ciertos procesos computacionales, véase (Durán, 2018; Durán and Formanek, 2018).

[23]Es importante señalar que la discusión sobre verificación de programas tiene su origen en el artículo de Richard A. De Millo, Richard J. Lipton y Alan J. Perlis (a DeMillo et al., 1979).

[24]Debemos notar que en la práctica los procesos de verificación y validación no son diferenciados analíticamente como lo hago yo aquí. Para una discusión a este respecto, véase (Morrison, 2015, Cap. 7).

[25]Una serie de cartas fueron enviadas al editor de Communications of the ACM Robert L. Aslzedzurst justo después de la publicación del artículo de Fetzer. Afortunadamente, las cartas fueron publicadas con una respuesta del autor, lo cual no solo mostró el espíritu democrático del

editor y de la revista, sino la reacción general de la comunidad con respecto a este tema.

Agradecimientos

La lista de agradecimientos incluye a Pío García, Marisa Velasco, Víctor Rodríguez, Julián Reynoso, Andrés Ilčić, Penélope Lodeyro, Maximiliano Bozzoli, Xavier Huvelle, y María Silvia Polzella. Muchas gracias por haber leído y comentado versiones anteriores de este trabajo, y por supuesto por el apoyo y cariño de todos estos años. Como parte de este trabajo fue realizado en Argentina, el CONICET y a la Universidad Nacional de Córdoba deben ser reconocidas por el apoyo económico brindado. Las instalaciones del CIFFyH (UNC) han sido la 'cocina' de muchas ideas que se incluyeron aquí. Muchas gracias a todos sus miembros por proveer una atmósfera tan (filosóficamente) estimulante. Finalmente, un agradecimiento muy especial a mi grupo de investigación "Simulaciones computacionales y experimentación desde la perspectiva de las prácticas científicas: una aproximación epistemológica y metodológica" PICT-2016-1524, FONCyT, CIFFyH, UNC, por el apoyo económico que permitió la publicación de este trabajo.

References

a DeMillo, R, R J Lipton, and a J Perlis. 1979. Social Process and proof of theorems and programs. *Communications of the ACM* 22 (5): 271–280. doi:10.1145/359104.359106.

Bidoit, Michel, and Peter D. Mosses. 2004. *CASL User Manual: Introduction to Using the Common Algebraic Specification Language*. Springer.

Bjorner, Dines, and Martin C Henson. 2007. *Logics of Specification Languages*. Springer.

Blanco, Javier, and Pío García. 2011. A categorial mistake in the formal verification debate. In *The computational turn: Past, presents, futures?*, eds. C. Ess and R. Hagengruber. Mv-Wissenschaft, Münster, Århus University..

Blass, Andreas, and Yuri Gurevich. 2003. Algorithms: A quest for absolute definitions. In *Bulletin of european associationfor theoretical computer science*, 195–225. https://www.microsoft.com/en-us/research/publication/164-algorithms-quest-absolute-definitions/.

Blass, Andreas, Nachum Dershowitz, and Yuri Gurevich. 2009. When are two algorithms the same? *The Bulletin of Symbolic Logic*.

Cantwell Smith, Brian. 1985. The Limits of Correctness. *ACM SIGCAS Computers and Society* 14 (1): 18–26.

Chabert, Jean-Claude, ed. 1994. *A History of Algorithms. From the Pebble to the Microchip*. Springer.

Dijkstra, Edsger W. 1974. Programming as a discipline of mathematical nature. *American Mathematical Monthly* 81 (6): 608–612.

Durán, Juan M. 2017. Varying the explanatory span: scientific explanation for computer simulations. *International Studies in the Philosophy of Science* 31 (1): 27–45. doi:10.1080/02698595.2017.1370929.

Durán, Juan M. 2018. *Computer simulations in science and engineering. concepts - practices - perspectives*. Springer.

Durán, Juan M., and Nico Formanek. 2018. Grounds for trust: Essential epistemic opacity and computational reliabilism. *unpublished*.

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

Fetzer, James H. 1988. Program verification: The very idea. *Communications of the ACM* 37 (9): 1048–1063.

Feynman, Richard P. 2001. *What do you care what other people think?* W. W. Norton & Company.

Hill, Robin K. 2015. What an Algorithm Is. *Philosophy & Technology* 29 (1): 35–59. doi:10.1007/s13347-014-0184-5. http://link.springer.com/article/10.1007/s13347-014-0184-5%5Cnhttp://link.springer.com/content/pdf/10.1007%2Fs13347-014-0184-5.pdf.

Hoare, C. A. R. 1971. *Computer Science*. New Lecture Series 62.

Hoare, C. A. R. 1989. *Essays in computing science*, Vol. 33.

Hoare, C. A. R. 1996. The Role of Formal Techniques: Past, Current and Future or How Did Software Get so Reliable without Proof? (Extended Abstract). In *18th International Conference on Software Engineering, Berlin, Germany, March 25-29, 1996, Proceedings*, eds. H. Dieter Rombach, T. S. E. Maibaum, and Marvin V. Zelkowitz, 233–234. IEEE Computer Society.

Hoare, C. A. R. 1999. A theory of programming: Denotational, algebraic and operational semantics, Technical report, Microsoft Research.

Humphreys, Paul, and Cyrille Imbert, eds. 2012. *Models, simulations, and representations. Routledge studies in the philosophy of science*. Routledge.

Humphreys, Paul W. 2004. *Extending ourselves: Computational science, empiricism, and scientific method*. Oxford University Press.

Moor, JH. 1978. Three Myths of Computer Science. *The British Journal for the Philosophy of Science* 29 (3): 213–222.

Morrison, Margaret. 2015. *Reconstructing Reality. Models, Mathematics, and Simulations*. Oxford University Press.

Newman, Julian. 2015. Epistemic opacity, confirmation holism and technical debt: Computer simulation in the light of empirical software engineering. In *History and philosophy of computing – third international conference, hapoc 2015*, eds. F. Gadducci and M. Tavosanis, 256–272. Springer. Springer.

Pfleeger, Shari Lawrence, and Joanne M. Atlee. 2009. *Software Engineering: Theory and Practice*. Prentice Hall.

Primiero, Giuseppe. 2016. Information in the philosophy of computer science. *The Routledge handbook of philosophy of information*.

Rapaport, William J. 1999. Implementation is semantic interpretation. *The Monist* 82 (1): 109–130.

Schembera, Björn. 2017. The science and art of simulation i, eds. Michael Resch, Andreas Kaminski, and Petra Gehring, 51–66. Springer.

Spivey, J. M. 2001. *The Z Notation: A Reference Manual*. Prentice Hall.

Turner, Raymond. 2011. Specification. *Minds and Machines* 21 (2): 135–152. doi:10.1007/s11023-011-9239-x.

Turner, Raymond. 2014. Programming languages as technical artifacts. *Philosophy & Technology* 27 (3): 377–397.

Turner, Raymond, and Nicola Angius. 2017. The philosophy of computer science, Spring 2017 edn. In *The stanford encyclopedia of philosophy*, ed. Edward N. Zalta. Metaphysics Research Lab, Stanford University.

Woolfson, Michael M., and Geoffrey J. Pert. 1999. *An introduction to computer simulations*. Oxford University Press.