Philos. Technol.
DOI 10.1007/s13347-014-0184-5

RESEARCH ARTICLE

What an Algorithm Is

Robin K. Hill

Received: 25 August 2014 / Accepted: 22 December 2014

Abstract The algorithm, a building block of computer science, is defined from an intuitive and pragmatic point of view, through a methodological lens of philosophy rather than that of formal computation. The treatment extracts properties of abstraction, control, structure, finiteness, effective mechanism, and imperativity, and intentional aspects of goal and preconditions. The focus on the algorithm as a robust conceptual object obviates issues of correctness and minimality. Neither the articulation of an algorithm nor the dynamic process constitute the algorithm itself. Analysis for implications in computer science and philosophy reveals unexpected results, new questions, and new perspectives on current questions, including the relationship between our informally construed algorithms and Turing machines. Exploration in terms of current computational and philosophical thinking invites further developments.

Keywords Algorithm · Philosophy of computer science · Church-Turing thesis · Mathematical ontology

1 Introduction

Computer scientists, academics who follow or use technology, and interested laypeople know that the algorithm is essential to computer work—the algorithm is the thing that programs implement, the thing that gets data processing and other computation done. Computer scientists, furthermore, know a set of algorithms by name, quite well, deploying the most appropriate for different tasks in different contexts. Let us

R. K. Hill (✉)
Department of Philosophy, University of Wyoming, Laramie, WY, USA
e-mail: hill@uwyo.edu

Published online: 11 January 2015

examine these objects, the algorithms, from an intuitive point of view rooted in this practice, starting with a look at three algorithms for the task of sorting.

Suppose we want to sort a set of index cards that are given to us one by one. Perhaps they represent (unique) postal or zip codes, or telephone area codes. We could sort them into numeric order (ascending) by placing each one, as it is given, into the stack we are holding by looking through the already-sorted stack until we find the right place, understood to be the point where the previous one is before the new one in the alphabet, and the next one is after. This is an Insertion Sort.

Suppose now that we face a similar task under slightly different circumstances, where we are sorting the index cards after they are spread out on a table. We might find the first in numeric order, pick it up; then find the next, pick it up and place it at the end of the stack in hand; and so forth. This is a Selection Sort.

And now consider yet a different method of arranging these numbers in order. We set up an array of switches, indexed from the lowest value (0) to the highest possible value (99999 for zip codes, 999 for area codes), and, for each value on the card, we flip on the switch with that index. When all the cards have been examined, we go through the array and emit a list of all the indexes where the switch is on, yielding a list of values that are present, in numerical order. This is a Bitmap Sort.

Three distinct methods have been described, all "effective," for putting a collection of values into order. All of these can be programmed and deployed to perform data processing tasks—and are, many times over, every year, every week. But what are "these?" The procedures that go by these names—Insertion Sort, Selection Sort, Bitmap Sort—along with a legion of other useful tools, are *algorithms*, the procedural building blocks of computer programming, and of general problem-solving in the realm of discrete operations and digital data. What is the nature of these objects, the algorithms?

2 The Question

2.1 Stakeholders

Because algorithms are important to our technological age, the educated person on the street might be expected to understand the nature of the algorithm, or at least to have formulated questions about it. Yet we see evidence that any procedure or decision process, however ill-defined, can be called an "algorithm" in the press and in public discourse. We hear, in the news, of "algorithms" that suggest potential mates for single people and algorithms that detect trends of financial benefit to marketers, with the implication that these algorithms may be right or wrong, rather than appropriate or inappropriate, relevant or misguided, in their application by the responsible party. A college student studying algorithms may be asked, by a relative at a family dinner, "Are you figuring out how to track me?" Most of those students would be taken aback, as the concern expressed is more pertinent to the content of a

set of rules and conditions, or to a data structure object, than it is to an algorithm as such. So just what are "algorithms as such?"

Because philosophers are interested in the status of mathematical objects like numbers, sets, proofs, and other abstract entities, they might be expected to have cultivated a consensus on the nature of the algorithm, a tantalizing and complex object. But whereas many philosophers are interested in algorithms as they relate to cognition, especially in the philosophy of mind, few have shown interest in the object per se.

Because algorithms are the raw material of programming, computer scientists might be expected to have a well-considered idea of the ontological status of the algorithm antecedent to the standard formalisms such as Turing machines, or at least to have articulated questions pertaining to that status. To the contrary, we have the editor of computer science's most prominent professional publication asking the question "What Is An Algorithm?" just recently and noting the lack of satisfactory consensus (Vardi 2012).

2.2 Current Answers

Moshe Vardi, that editor (of the *Communications of the ACM*), described two different formal approaches (Vardi 2012), the recursors of Moschovakis and also the abstract state machines of Gurevich, and suggested that both constitute definitions of "algorithm," and that they might even be logically equivalent (Moschovakis 2002; Gurevich 2011). According to Moschovakis, an algorithm is a "defineable recursor," a partially-ordered domain set with a transition mapping to determine states, and a value mapping to determine results. According to Gurevich in an earlier paper (Gurevich 2000), an algorithm is an object that satisfies sequential-time, abstract-state, and bounded-exploration postulates. Cogent as these formalisms may be, this kind of definition is akin (as Moschovakis himself mentions) to telling people that the number 2 is the set $\{\emptyset, \{\emptyset\}\}$; it is likely to leave the interested public less, rather than more, enlightened. In response, I wrote a letter to the editor pleading for an account less formal.

> A proper definition would allow computer scientists to talk to the public about algorithms. The pragmatic motivation is to justify the treatment we informally apply to algorithms: construct, explain, and exchange; admire for elegance; debate whether one is the same as another; marvel that the one for the program Eliza is so simple; patent or sell them; or preclude protections based on principled argument. The theoretical motivation is to probe relationships to programs and abstract machines and promote discussion of semantics and implementation (Hill 2013).

The search for that proper real-world definition inspires this paper. Civilization needs informed policy-makers, its citizens able to ask, "Where does the PageRank

algorithm, used in Google search, get the base authority metric?" What would enable those citizens to ask that question is the understanding that an algorithm dependent on built-up authority values must start with some particular numbers. As software grows ever more complex, stakeholders, managers, and legislators must be able to discuss algorithms with software engineers, confident that their exchange is grounded in mutual understanding.

2.3 Classic Answers

Classic definitions of "algorithm" are often embedded in the mathematical discourse that formed inchoate computer science. Common elements include deterministic or effective rules or procedures, and computation on inputs to produce outputs. Rapaport (2012) gives a good overview (in his Appendix, entitled "What is an Algorithm?") of several foundational definitions in a treatment that invites broader consideration. According to Markov and Nagorny via the Greendlinger translation, "An algorithm is a prescription uniquely determining the course of certain constructive processes (Markov and Nagorny 1988)."

Kleene's informal starting point describes an algorithm as a procedure such that:

> If (*after* the procedure has been described) we select *any* question of the class, the procedure will then tell us how to perform successive steps, after a finite number of which we will have the answer to the questions we selected. In performing the steps, we have only to follow the instructions mechanically, like robots; no insight or ingenuity or invention is required of us. After any step, if we don't have the answer yet, the instructions together with the existing situation will tell us what to do next. The instructions will enable us to recognize when the steps come to an end, and to read off from the resulting situation the answer to the question, "yes" or "no". In particular, since no human performer can utilize more than a finite amount of information, the description of the procedure, by a list of rules or instructions, must be finite. (Kleene 1967, page 223; italics his)

He goes on to describe the algorithm first in terms of a decision procedure, and then a computation procedure, introducing formal definitions to show that they are equivalent, and finally settling on the Turing machine for his exposition. Marvin Minsky's contemporaneous book takes a slightly different path to a similar result. His definition that "an effective procedure is a set of rules which tell us, from moment to moment, how to behave" (Minsky 1967, page 106) is compatible with the others. His discussion addresses how to secure our confidence that a given set of rules is effective, ultimately observing that specification of the mechanism that is to interpret those rules would suffice, which leads to the universal Turing machine.

Knuth says that an algorithm is "a finite set of rules that gives a sequence of operations for solving a specific type of problem," with five additional important features—finiteness, definiteness, input, output, and effectiveness (Knuth 1997). Rapaport himself synthesizes the various historical contributions this way:

An algorithm (for executor E to accomplish goal G) is:

A procedure (or method)—i.e., a finite set (or sequence) of statements (or rules, or instructions)—such that each statement is:

- Composed of a finite number of symbols (or marks) from a finite alphabet
- And unambiguous for E—i.e.,
  - E knows how to do it
  - E can do it
  - it can be done in a finite amount of time
  - and, after doing it, E knows what to do next—
- And the procedure takes a finite amount of time, i.e., halts,
- And it ends with G accomplished.

(Rapaport 2012, Appendix)

Both Moschovakis and Gurevich themselves suggest that there is something to "algorithm" beyond symbolic manipulation. Moschovakis says that the proper role of a definition is to identify and delineate fundamental mathematical properties, and suggests that algorithms embody something above and beyond that, namely, the "purposeful interpretation of equations" (Moschovakis 2002, page 5). According to Gurevich in his critique of Moschovakis (Gurevich 2011, page 13), the meaning of the recursor is nothing beyond the least fixed point construction, which presumably does not suffice to capture the full connotations of *algorithm*, but he agrees that it is useful for mathematical analysis. Their acknowledgement of something significant beyond the formal definition is unusual among contemporary computer scientists, but in 1979, V.A. Uspensky and A.L. Semenov made this point at a symposium on the subject of algorithms:[1]

> The most important discovery in the science of algorithms was undoubtedly the discovery of the general notion of algorithm itself as a new and separate entity. We emphasize that this discovery should not be confused with representational computational models (constructed by Turing, Post, Markov, Kolmogorov)... we are of the opinion that these constructions were only introduced in order to provide a formal characterization of the informal concept of algorithm (Uspensky and Semenov 1981, Page 104).

Sieg (2008) takes on the problem of bringing the formal concept into the real world by providing axioms of boundedness and locality meant to manifest physical system properties; those axioms, of course, are part of a formal definition. Certainly, explicating algorithms in terms of abstract machines, or other formal constructs, is a worthy effort, and my intention is to complement rather than contradict it.

[1] I am indebted to Ksenia Tatarchenko, who introduced me to the work of the 1979 symposium *Algorithms in Modern Mathematics and Computer Science* in her talk at the 2014 conference of the International Association for Computing and Philosophy.

The discussion will appeal to the philosophy of mathematics for guidance. Philosophical inquiry unapologetically relies on intuition and quotidian happenstance in order to probe ontology. In the case of mathematics, for instance, our successful and collaborative use of numbers is a given, stipulated without controversy, affording a foundation upon which philosophers can build. In logic, how we recognize the correctness of a logical conclusion is a knotty problem, but the fact that ordinary people do recognize validity, and perform other feats of logic, quite often and quite accurately, is not under dispute. In the context of mathematical logic, Kreisel (1967) defends the "old-fashioned idea" that "one obtains rules and definitions by analyzing intuitive notions and putting down their properties." Let us follow his example in seeking a definition that is both more specific than the general classical statements and less technical than the current formal views.

What is an algorithm, as we name, explain, teach, and use it? Let us abandon the prevailing premise that what a computer *does* must inform what an algorithm *is*. Let us start from the other side. Hence, the title of this paper, in its colloquial grammar, displays a non-formal approach.

3 The Setting

3.1 The Materials of the Inquiry

For those who have not acquired a strong sense of what constitutes an algorithm, the following examples should introduce the idea and at the same time, persuade the reader that the algorithm really is a robust concept. Here are some of computer science's favorite named algorithms, taught in undergraduate computer science. We limit the scope to relatively simple single-thread algorithms in order to sharpen the focus. (This paper does not consider parallel, quantum, hypercomputational, or other algorithmic possibilities.) In the popular press, MacCormick (2012) gives a nice overview of significant modern algorithms for the layperson.

These algorithms are given as capsule vernacular descriptions supplemented with structured pseudocode, a combination that is intended to be both sufficiently accessible and sufficiently precise for a variety of readers, and to avoid format or implementation assumptions. These descriptions are glosses; this work makes no claim regarding a canonical expression of an algorithm. Symbols, even variables, are minimized, but cannot be eliminated completely. Comments, not part of the algorithm, but mere explication of detail, are enclosed in "/*" and "*/." Although this yields descriptions that anachronistically resemble COBOL, I equivocate on what the expression of an algorithm looks like in the hope that a loose written presentation will quash any tendency to attribute algorithm-hood based on form alone. The reader should not be led to the conclusion that any sequence of nicely-indented statements starting with verbs constitutes an algorithm.

3.2 Three Common Sorting Algorithms

Let us start our examples with the sorting algorithms already described, which arrange values in some predefined order.

Insertion Sort For each value X from the unsorted list, go through the sorted list until the current value is less than X and the next value is greater than X, and put X after the current one.

```
SET Length of SortedList to 0
WHILE InputList not empty
  READ next value X from InputList
  SET Index to 0
  /* Find proper place for X */
  WHILE (X < SortedList at Index) AND (Index < Length)
    ADD 1 to Index
  ENDWHILE
  SET Placement to Index-1
  ADD 1 to Length of SortedList
  SET Index to Length of SortedList
  /* Move other values to make room */
  WHILE Index >= Placement
    SET SortedList at Index to SortedList at Index-1
    SUBTRACT 1 from Index
  ENDWHILE
  /* Put X in place */
  SET SortedList at Placement to X
ENDWHILE
```

Selection Sort Find the next-highest value in the unsorted set of size `N`, and place it at the end of the sorted list.

```
SET SortedCount to 0
  WHILE SortedCount <= N
    SET Index to 0
    SET MinPosition to 0
    /* Find next smallest value */
    WHILE (Index < N)
      IF InputList at Index < InputList at MinPosition
        SET MinPosition to Index
      ENDIF
      ADD 1 to Index
    ENDWHILE
    /* Output it */
    WRITE InputList at MinPosition
    SET InputList at MinPosition to MaxValue
    ADD 1 to SortedCount
  ENDWHILE
```

Bitmap Sort Input each integer in a list of N and set the array location indexed by that integer to True; then for each array location, in order, output the index of that location if the value is True.

```
SET Index to 0
/* Initialize Detection_array to all False */
WHILE Index < Length(Detection_array)
  SET Detection_array at Index to False
  ADD 1 to Index
ENDWHILE
SET Index to 0
/* For each value in InputList, set its place to True */
WHILE Index < N
  SET Detection_array at (InputList at Index) to True.
  ADD 1 to Index
ENDWHILE
SET Index to 0.
/* Produce a list, in order, of all values present */
WHILE Index < Length(Detection_array)
  IF Detection_array at Index
    PRINT Index
  ENDIF
  ADD 1 to Index
ENDWHILE
```

3.3 Other Algorithm Examples

In addition to sorting, another (perhaps *the* other) main data processing task performed by computers is searching. One algorithm (Sequential Search) for performing a search through a set of data is to look at each individual data value and compare it to the search target until that target is found. Let us examine a different search to augment our concept of the algorithm.

Binary Search Divide search space (a sorted list of size N) into halves, compare target value X with midpoint value, then narrow down search space to upper or lower half. Repeat until only one value, X, remains.

```
SET LOWER to 0
SET UPPER to N - 1
SET Found to False
WHILE LOWER < UPPER
  SET Midpoint to N div 2
  IF InputList at Midpoint > X
    SET Upper to Midpoint
  ELSE
  IF InputList at Midpoint < X
    SET Lower to Midpoint
  ELSE
    WRITE "Found at" Midpoint
    SET Found to True
  ENDIF
ENDWHILE
IF NOT Found
  WRITE "Not Found"
ENDIF
```

Note that Binary Search is easily illustrated by a non-digital manual search through a drawer of cards in a library's card catalog, or through a large telephone book (with both sorted in alphabetical order): We pick a card, or a page, about halfway through, and based on whether its value is greater or less than our target, we abandon either the upper or lower half of the search area (namely, card catalog or telephone book), and repeat the search operation on the remaining half, cycling until the target is found.

Lest we think that these algorithms are inextricably bound up with modern computers, here is one from the classical age of geometry that does not involve digital machines.

Bisecting an Angle Given an angle with two lines extending from a vertex, draw points at some arbitrary but equal distance from the vertex on each of the lines of the angle, then draw circles of equal size through each point, large enough to cross in the middle. Draw a line from the vertex through the points of intersection of the circles. That line will produce two new angles, each half the size of the original.

```
Measure arbitrary x from vertex A on Line 1, mark point B
Measure x from vertex A on Line 2, mark point C
Scribe a circle at an arbitrary point B that reaches more
    than half-way to Line 2
Scribe a circle, using the same setting, at point C
  Let D be the nearest point of intersection
  Let E be the farthest point of intersection
Draw a line from A through D and E
```

4 The First Cut at a Definition

While the preceding examples, along with the synthesized definition given by Rapaport in Section 2.3, should suffice to give non-technical readers the idea of the algorithm, we stipulate some of the details explicitly:

1. The steps are followed in the sequence given, unless a condition leads to different steps based on its results; in any case, what step to take next is unambiguous. Conditional steps are skipped if the condition does not apply.
2. The steps are discrete; the completion of a step is realized in some obvious way, and a step is not started until the current one is finished.
3. References are not ambiguous, and references to things with the same name are references to the same thing.
4. If a subprocedure is invoked, replacement of that subprocedure call with the complete subprocedure instructions adheres to these requirements.

Note the important distinction between an algorithm and a task. We have examined, for instance, the common task *sorting*, and we have seen three different algorithms that accomplish it. Algorithms can also accomplish more than one task. A sort on a field with a value shared by several entries also serves to group those entries. The grouping may be a completely different goal in the user's mind, constituting a task other than sorting. Properties of a data processing or computational task per se, such as sorting (which can be subdivided into comparison-based

and non-comparison-based, with Bitmap Sort in the latter category) is the subject of much study in computer science, but not here. The subject of this investigation is the algorithm, not the task.

We place considerable trust in these examples to drive the reader's conceptual apparatus in the right direction. A non-ostensive definition of *algorithm* is not adequate. Without illustrations, an intelligent layperson may wonder, "Why is this supposed to be interesting? It is just a set of instructions." The examples should convince the reader that an algorithm is quite a special—precise and incontrovertible—set of instructions.

4.1 The Definition, Preliminary

These algorithms are named structures rather than simple atomic objects, they provide control rather than data, they are finitely expressed in imperative terms, and they are effective in the sense of requiring no judgment or insight. We have seen nothing that contradicts any of the definitions, from experts throughout the history of computer science, already cited. My intention is to supplement rather than dispute those definitions, the informality making analytic comparison unproductive, and the differences reflecting emphasis rather than substance.

I propose this preliminary definition, to be refined on examination:

Definition 1 An *algorithm* is a finite, abstract, effective, compound control structure, imperatively given.[2]

This definition is closest, among the historical contributions, to that of Knuth (good company, although any of them would be). The history of computation theory (with which the reader is assumed to be familiar) can be seen as a struggle to isolate, identify, and describe these characteristics. The usual context is the Church-Turing thesis, which states that effective calculability in the practical sense, as instantiated by humans, is equivalent to Turing machine computability in the formal sense, as instantiated by digital computers. This paper attempts to shed light on the first referent, the non-formal sense of "effective calculability," which must be elucidated independently of the Turing machine or other equivalent formalism if the Church-Turing thesis is to be innocent of begging the question.

Let us elaborate slightly on the individual elements of the proposed definition. Understand that the subject here is the instructions, not their execution, a point addressed further in Section 5.5.

Finite: Allows a representation to be articulated in finite time and space.

[2] In my earlier letter to the editor (Hill 2013), I offered a definition that said "expressed in a finite, imperative form." This has turned out to be misleading; the expression is independent of the algorithm, as is discussed in Section 5.5. The letter also used the term "deterministic" rather than "effective."

Abstract: Has no space-time locus, allowing talk about an algorithm independently of instances. This means that an algorithm is general, applicable beyond a single specific instance of a task. Although the classic definitions do not stipulate "abstract," they all seem to assume it.

Effective: Requiring no judgment (except objective evaluation of conditions that may appear), learning, insight, or understanding; devoid of open questions; not subject to interpretation. All of the historical definitions incorporate this property, with the same word or others. In accordance with Rosser (1939), by "effective," I mean precisely predetermined and certain to produce the result. The term "mechanical" appeals, but mechanism is a subject of study in its own right in the philosophy of technology, so I use "effective" on the understanding that it incorporates the colloquial aspects of "mechanical."

Control: Supplying content that brings about some kind of change from one state to another, expressed in values of variables and consequent actions, broadly construed. The phrase "compound control structure" is used to avoid confusion with the simple "control structure" used in computer science pedagogy to denote a single step. The classic definitions capture this aspect in the stipulation of carrying out operations, instructions, rules, procedures, or prescriptions.

Structure: Organized, consisting of smaller units, in this case, steps under a partial order. (A partial order, rather than a linear order, allows that some steps may not have to be strictly ordered relative to each other. For convenience, the articulation of an algorithm gives steps in some particular sequence, even if arbitrary.) As noted above, these structures are compounds. We do not normally regard a single conditional, or a single loop, or a single assignment, to be an algorithm. In their definitions, Kleene, Minsky, and Knuth all imply a composition of smaller objects.

Imperative: A how-to; gives directions or orders. This aspect echoes the "prescription" of Markov's definition, the "instructions" of others, and the procedure that "will ... tell us how" of Kleene's definition; although not explicit, it is arguably implied by all.

For contrast, what things are close, but not quite there?

- Other abstract, finite objects include numbers, such as the natural numbers in the set $\mathbb{N}$. But they are not structures, under ordinary use and connotations.
- Other abstract compound structures include proofs. But they are not controls, nor imperatively given.
- Other imperative compound control structures include traffic signs and orders issued to soldiers. But they are not abstract.

5 A Refined Definition

5.1 Additional Aspects Revealed

No learner or prospective user is likely to be given an algorithm's instructions without some understanding of the task in question. From the human point of view, there

is way more to an algorithm than its procedure. Consider a couple of additional examples.

Rot13 Shift each letter in the plain text by 13 down the alphabet, wrapping around to the beginning when necessary, and write the new letter at that place.

This explication of an algorithm, given above following the earlier model, never states what Rot13 actually *does*, which is disguising text through a simple substitution cipher. It also falls short of capturing the interesting feature that the decoding is the same as encoding, due to the shift that takes a letter exactly half-way along the (English) alphabet. Such a self-inverse function (where two applications of the function to an argument yields the original value), obviating the need for a separate decoding procedure, may be part of the goal.

Eliza Match patterns in the input text to select actions or repeat key input phrases embedded in questions or prompts in the output text.

The early artificial intelligence program ELIZA, in its DOCTOR script, carried on an interaction with a human participant that was intended to mimic Rogerian therapy (Weizenbaum 1966).

> When the "patient" exceeded the very small knowledge base, DOCTOR might provide a generic response, for example, responding to "My head hurts" with "Why do you say your head hurts?" A possible response to "My mother hates me" would be "Who else in your family hates you ?"(Wikipedia 2013)

The definition falls short because a significant feature of this algorithm—what makes it famous—is its simplicity, and we cannot appreciate that without grasping its purpose. (This account skirts an issue; some would say that the algorithm is actually "pattern matching." I appeal to historical use in calling the whole procedure "Eliza.")

And what about the fact that Bitmap Sort, introduced as a parallel to the other sorting algorithms, requires a direct-access data structure indexed with all possible data values and initialized throughout to "False"? And that Binary Search carries the special requirement that the data be sorted in order first? Our point of view, which takes practice as primary, justifies taking these factors to be integral parts of the algorithm itself.

5.2 The Definition, Augmented

In light of these additional aspects of the human perspective on algorithms, we need another compound criterion:

Provisions and Purpose: (1) Accomplishing a given purpose (2) Under given provisions.

This looks a lot like a specification, as presented by Turner (2011), which provides (for a program or software system) the measure of correctness or malfunction. Surely that belongs in this picture, and surely specification is what we intend. Because specification is often treated formally, however, we will stay with the mundane terms

given. In effect, the refinement to the definition manifests the intentionality of algorithms, which they share with specifications. It also provides a contrast with recent work on mechanisms, as synthesized and analyzed by Piccinini. He says (Piccinini 2007, page 502) that "the mechanistic account does not appeal to semantic properties to individuate computing mechanisms and the functions they compute." In our opinion, however, algorithms are all about semantics, with "what it means" taken to be "what it does" for us.

It seems prudent to treat this extra criterion independently. We retain the first definition, Definition 1, as a fallback if and when these additional qualifications become burdensome. Our new and refined definition reads thus:

Definition 2 An *algorithm* is a finite, abstract, effective, compound control structure, imperatively given, accomplishing a given purpose under given provisions.

The nature of the additional aspects, in particular the ontology of the purpose and provisions, may be troubling. The provisions or context, and the result, purpose, or task accomplished, are plain and explicit descriptions of states, and therefore declarative. So be it, as long as we can accept the formulation of those states in our imperative framework. For a homogeneous concept, a possible accommodation is to think of those descriptions as "ifs," the provisions as preconditions and purpose as postconditions. But we can also just let it lie, without any need to specify further, since we do not commit to any particular algorithmic expression or formulation.

So here is a selection from our stable of examples, modified to include *Provisions and Purpose*, in their capsule vernacular forms (without the pseudocode-like instructions).

Insertion Sort To produce a sorted output list from an unsorted input list, take each value X from the unsorted list, go through the sorted list until the current value is less than X and the next value is greater than X, and put X there.

Selection Sort To produce a sorted output list from an unsorted input list, find the next-highest value in the unsorted set, and place it at the end of the sorted list.

Binary Search To find a value X in a sorted array, divide search space into halves, compare target value X with midpoint value, then narrow down search space to upper or lower half. Repeat until only one value, X, remains.

Bisecting an Angle To find the line that bisects a given angle, draw points at some arbitrary but equal distance from the vertex on each of the lines of the angle, draw an arc through each point centered on the apex, and then draw a line from the apex through the intersection of the arcs.

Rot13 To encrypt a plaintext into a form that cannot be easily understood on casual reading, such that no extra decoding mechanism is necessary, shift each letter in the plaintext by 13 down the alphabet, wrapping around to the beginning when necessary, and write that code letter in its place in the cyphertext.

Eliza To simulate sympathetic human conversation, match patterns in human speech input to select actions or repeat key input phrases, embedded in questions or prompts for more.

5.3 Effects of the Proposed Definition

The proposed Definition 2 includes compass-and-straightedge algorithms, to bisect an angle, for instance, and other non-electronic but straightforward sets of instructions such as those (ideally) for IRS tax forms. Note that the *Provisions and Purpose* cannot be simply extracted from what an algorithm happens to do, retroactively. In common with ideal specification (which is, however, often modified under the pressure of development, as noted by Turner 2011), and in accordance with Kleene's requirement given in Section 2.3, the procedure must be known in full before its deployment. There are no circumstances in which an abstract compound control structure appears on our list before its goal and conditions are understood.

5.3.1 Some Other Compound Control Structures

Consider a few other well-known processes, as follows. Are these things merely close to our definition of "algorithm," or are they subsumed by it?

Game of Life (Conway's) In this engaging iterative display, the application of primitive rules, to a small initial matrix of simple cells, for propagation or annihilation of cell presence, produces configurations and patterns (Gardner 1970). Demonstration and particular patterns can be found online at many websites, such as Wikipedia.

Recipe for Steamed Pudding Here's one: mix together, in the order given, 1 cup flour, 1–1/2 teaspoons baking powder, 1–1/2 teaspoon salt, 1/3 cup brown sugar, 1/2 cup bread crumbs, 2/3 cup finely chopped suet, 1 cup chopped cranberries, 1 egg, 1/3 cup milk or water. Turn into a well-greased mold. Cover with wax paper, and steam for 2 h.

Recursive Definition of Factorial This definition is the classic way to give the value of the nth factorial, which is the product of n and all the positive integers less than n.

$$factorial(0) = 1$$
$$factorial(n) = n \times factorial(n-1)$$

Checkers A board game that follows rules given sketchily as follows, and more fully at, for instance, GameTable Online Inc. (2013).

Checkers is played on a standard 8×8 board of alternating colors (dark and light). Each of two players starts with 12 pieces set on alternating squares. The player with the darker colored checkers goes first; thence, the players take turns after that. In a turn, the current player must move one of their checkers if they can. Checkers move one space diagonally forward into an un-occupied location... (and so forth).

Each object given above is easily seen to be an abstract compound control structure. Each is a general procedure, with organized steps, in which the content is control, applicable to different circumstances. Is it an algorithm under the definition?

Recipe? The definition excludes recipes, which are not effective except in perversely rigorous cases. Many everyday procedures fall into this category, even those that seem to have solid and straightforward instructions, such as changing a tire and shucking peas. They involve adequacy judgment, for example, on acts such as "tighten the lug nuts"; they involve unstated assumptions, for example, that a pod empty of all but tiny shriveled peas signifies a termination condition. The learning and enactment of those procedures admits of mistakes, training, and improvement, while the mental grasping of an algorithm is discontinuous. If a set of instructions can be said to be followed *well* or to be followed *badly*, it is not an algorithm.

This view stands at cross-purposes to the popular use of the recipe analog for introducing the notion of an algorithm. In fact, Cleland (2001) claims that a recipe (as opposed to a Turing machine) is indeed exactly an algorithm because the physical consequences of its actions are explicated in the instructions, but my view places a greater emphasis on mechanics.

Game of Life and Checkers? The definition excludes games, which do not offer an imperative compound control structure that is guaranteed to terminate in the accomplishment of a pre-set goal. A two-player game is interactive, and conceptually uncertain (else there would be no reason to play). Of course, a game, in the sense of a sequence of moves, can be written as a set of conditionals and actions to further the goal of winning under every possible move made by other players, but that seems to place it outside the normal conception of a game, where the common view of the whole situation involves an element of surprise. In fact, the algorithm thus described might better be called "Exhaustive Search" than, say, "Checkers."

In the case of the Game of Life, no termination is built in. Certainly, algorithms are available to produce certain cell patterns in the Game of Life; these subsets of instructions fit our definition because they terminate in a certain pre-set goal, while the Game of Life process (presumably) continues.

Recursive Definition of Factorial? And, perhaps most controversial, this view of *algorithm* excludes recursive definitions, which are not imperative. A definition of the factorial function, such as that above, with a base case and a recursive case, does not undertake a computation of a factorial nor does it provide the directions. Such a definition is a declarative. An algorithm must "do," not "be." As Gurevich (2000, p. 10) points out, one possible algorithm generally available via such a definition starts with "Apply the equations..." In other words, one possible algorithm tells us the obvious thing to do with the definition, but that outstrips the definition itself.

This last point deserves more discussion, as follows.

5.4 The Place of the Imperative

Unlike the classic definitions (although some suggest it), our definition treats the imperative feature as an essential characteristic of an algorithm—not the imperative voice, or the imperative mood, which are artifacts of the written expression of an algorithm, but imperativity in the sense of a core property. To explain an algorithm via

a natural language, we ultimately cast its procedural quality as imperative utterances. In the case of Bitmap Sort, we can give a list of unsorted integers, and define the initial state of an array, indexed by the range of possible integers, as False everywhere. We can define the resulting state of that array as carrying the value True in every location where the index is an integer found in the unsorted list. Yet this description in terms of assertions still does not tell us exactly what to *do*. An astute student would figure it out, but the "it," the algorithm, still must be specified. In the case of Binary Search, we can define the initial circumstances as the sorted dataset and the target value. We can define the test of comparison, and the set of subsequent steps from the different conditions that might hold. These are the assertions contained in a recursive definition. Again, for a clever beginner, this might be enough to construct an algorithm in the head, so to speak, but it is not the algorithm itself.

Until we issue a command, or order an action, we have not conveyed an algorithm. Although the articulation is not the algorithm itself, the algorithm itself being the procedure, the only means available to us of articulating the procedure is the imperative verb. That is why Definition 2 says "imperatively given." Some might claim that an algorithm can be cast as a narrative, as in "The computer [or 'the algorithm'] splits the search space array by dividing the index in half to get the next index, then it compares the value at that index to the target value..." This, however, is an indirect rendition that must assume, identify, and incorporate an agent, superfluous to the algorithm itself. Note that Rapaport's synthesized definition does so (Section 2.3), but we take it to be describing the setting as well as the algorithm.

Cleland makes a similar claim for the connection between her instruction-expressions and action-types:

> ...the instruction-expressions literally order the performance of the action-types they designate; I use the expression "prescribe" to distinguish this special referential relation between an instruction-expression and an action-type from other referential relations (Cleland 2001, page 221).[3]

Gurevich (2011, p. 13) gives two brief but suggestive comments in quick succession, juxtaposed here: "... recursor theorists underestimate the abstraction capabilities of imperative programming" and "[The declarative nature of recursion] is by itself a limitation for software specification..." Although these remarks are given in the context of programming, which is the computer version rather than the quotidian version of the algorithm, they suggest that the declarative form is inadequate for expressing an algorithm. In short, according to the earlier paper of Uspensky and Semenov (1981, p. 100):

> The meaning of a term or formula is indicative; a term indicates a thing, a formula–a fact. The meaning of an algorithm is imperative: an algorithm is to be performed.

[3]Note that although Cleland uses the word "order" a few lines later as a noun to mean "correct sequence," she seems to be using "order" in this passage to mean "command," referring to the imperative.

Note that data structures, which are non-procedural components, still play a critical role. The "cleverness" of Bitmap Sort is in the array of boolean, or rather, in the idea of using an ordered array of boolean to track the presence or absence of an entry rather than an array of values to track the entry's place in the ordering relative to others. But we still need the imperative description, however modest it may be. A parallel study of data structures might confer a similar (but declarative) ontology on them. The programming construct of choice is often the *class*, or abstract data type, a combination of data structures and associated algorithms working together, with the preconditions and postconditions viewed as the assertions that come from formal verification. The unit of solution, so to speak, in computer science is the abstract data type. But even under those circumstances, the choice of algorithms remains open. An abstract data type for strings is quite likely to include a search, but many different search algorithms are candidates. In any case, we must allow that algorithms and data structures work closely together, if the concepts are to reflect the real world, though we will not foresake our commitment to the algorithm as an object in its own right.

A philosophical perspective might suggest that the imperative quality could be factored out into causality, which is then imposed on a recursive definition to wrap it all up tidily. But an explicit choice of method is still necessary. In fact, an algorithm is exactly that—a method subject to explicit choice. The difference between a declarative definition and an algorithm is akin to the difference between a non-procedural geographic specification, as in the street address of a house, and the procedural geographic specification, as in the travel directions to that house. Both are useful, but the former cannot simply be finessed into the latter. Only the latter can be an algorithm.

5.5 Exploration

To the careful observer, what might this definition lack? Some might call for a "correctness" property, but it should be obvious, on reflection, that such a criterion would be redundant. An algorithm for x really performs x. We are not checking programs against some ideal; we are talking about the ideal itself. Other additional properties might seem reasonable, however.

Minimality If we were presented with an algorithm that contained extraneous steps, we would be inclined to remove them (If we see a set given as $\{j, e, w, w, k, x, w\}$, we would be inclined to correct it to $\{j, e, w, k, x\}$). Should we require an algorithm to be minimal in its expression? Here, "minimal" does not mean the briefest set of instructions that accomplishes the task—that might constitute a different algorithm altogether, a more clever one, taking a more direct approach—no, here, "minimal" means free of irrelevant digressions and steps that have no effect on the main course of the procedure. That criterion is meaningful when applied to programs, but to apply it to the algorithm confuses the algorithm with its representation. The algorithm itself has no competing extensions, some larger and some smaller. It is already minimal, just as it is already correct.

Compositionality Can algorithms contain other algorithms? Many respectable named algorithms contain a search, just as many contain a sort, as a subprocedure. The answer, then, is yes, algorithms are compositional; not only can they be

plugged together, in sequence, subject to compatibility of inputs and outputs, but they can be nested in a hierarchy that accomplishes a single goal. But the concept calls for judicious thrift. I appeal to software engineering practice, and observe that we would not call a complete point-of-sale system an algorithm, nor a flight simulator, nor a memory-block indexing system. They are not only continuous in operation, but too complex; the references do not conjure specific methods that terminate at a goal. These systems are compounded of implementations of underlying algorithms.

Static Versus Dynamic Manifestation One might ask whether our "algorithm" is static or dynamic, or both. The answer is: static. What we have in mind is the set of instructions, not their execution. In terms of ontology, this is similar to a program and a process, or a song and its performance; our focus is on the song rather than the performance. Note that either an algorithm or its execution can be sold and traded, the static version as code, the dynamic version running in some hardware. But we can distinguish between those reasonably well. The dynamic object is an application, or execution, of some implementation of the algorithm, and it has properties beyond those given in our definition. The dynamic object may not be finite, for example. But in our algorithm, our set of instructions, there must be a termination that will apply in the given conditions in order to accomplish the task, according to the *Provisions and Purpose* criterion. Algorithms may fail in execution, but it is "not their fault." What has failed, in running infinitely or yielding an incorrect result, is a flawed implementation or application.

Written Versus Essential Form An algorithm can be written down in brief vernacular explanation, or pseudocode—both used here—or as a flowchart, or a program, or a circuit. This discussion relies heavily on that written form, but it is not the algorithm itself. Cleland (2001), again, makes an analogous point in the context of her thesis: "It is important to distinguish instruction-expressions from instructions. For the identity of a quotidian procedure depends upon the identity of its constituent instructions, not the vehicle of their expression." So many written forms, in different languages and notational devices, may obscure the nature of the object. For one thing, the written forms provide the instructions in linear order, as mentioned, whereas the fundamental organization is actually a partial order. Suffice it to say that this investigation focuses on the *object behind* the several written forms that we would agree capture the same algorithm, the object hovering somewhere between the sketchy vernacular description and the complete programmatic implementation.

Is the ontology of the algorithm a subject worth investigating? Many happenings of ordinary life lack the weight to exercise our ontological curiosity. What is the ontology of a drive in the country? Of a facial expression? Of a sale of goods? What is the ontology of a recipe? We may, upon occasion, care whether one recipe for goulash is the same as another, or whether we have a favorite drive in the country (and whether that constitutes one historical event or a general route), but hardly ever do we care in a rigorous fashion. An algorithm, however, is distinct and distinguished enough to bear and repay scrutiny. A conversation is a structured sequence, but we do not give names to particular conversations, and thereby confer identities upon them.

We do not press them into service in other settings, and thereby lend purposes to them. But, with algorithms, we do.

6 The Computational Position

6.1 Tradition: Favoring the Formal

In computer science, practitioners describe, refine, encode, and exchange algorithms among themselves, while theorists measure, classify, and analyze algorithms. The concept *algorithm* carries deep significance, and many computer scientists are assured that they already know what an algorithm is—it is a Turing machine. This is reasonable on the face of it, as a Turing machine is a robust and intuitively appealing construct, representing a general notion of "how to do" some task with a computer, and relentlessly mechanical. Digital computers explicated by formal methods are the basis of virtually all current questions in the philosophy of computer science. To wit, the entry in the *Stanford Encyclopedia of Philosophy* for "Philosophy of Computer Science" lists the topics that it addresses: "specification, implementation, semantics, programs, programming, correctness, abstraction, and computation" (Turner 2013). These are all formal, and legitimate, subjects of research, posing meaningful questions. A formal system is a thing of beauty, and algorithms are formal objects *in some important sense*, just as the number 2 is $\{\emptyset, \{\emptyset\}\}$ in some important sense.

But when we teach algorithms, we do not merely hand our students Turing machines, or any of the well-known equivalent formalisms. A Turing machine M is a 7-part compendium, according to the formal definition in the classic textbook by Hopcroft and Ullman (1979, p. 148): $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, with Q a set of states, Γ a set of symbols with Σ a set of input symbols, q_0 a distinguished start state, B a distinguished "blank" symbol, F a set of final states, and the transition function δ a mapping from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$, giving the set of next steps, one for each reachable state. This means that a Turing machine consists of several bunches of symbols and quintuples. It does not do anything, nor purport to direct any action, nor even suggest it. As stated by Dodig-Crnkovic (2011) (italics hers), "*The Turing Machine essentially presupposes a human as a part of a system*—the human is the one who poses the questions, provides material resources and interprets the answers." We may wish to broaden this requirement to *agent*, human or not. Markov and Nagorny (1988, page 102) say that the word "prescription" in their definition of an algorithm "indicates the presence of an addressee—a subject able to fulfill this prescription. In particular, this can be a person or some good model of one." This suggestion that an agent is necessary reinforces the view that a Turing machine is inert.

While we may preach that an algorithm is a Turing machine, we do not practice it. When we grade a stack of programs that purport to implement Binary Search, we measure them against a robust idea that we call "Binary Search," not against any septuple that constitutes a Turing machine. In my thirty-odd years of teaching, I have never written (nor run, nor seen) a Turing machine for Binary Search, nor Bitmap Sort, nor any other full-fledged algorithm. The Turing machine is an essential part

of the curriculum, but its significance is taught by its general construction, execution principles, and tiny examples. To students, I teach an algorithm via vernacular, just as in this paper, and hand-waving and demonstration, until I see the light dawn. The programming exercise that follows draws it more sharply and corrects misunderstandings. Given the sketchy description of Binary Search or the other algorithms offered earlier, a critic may point out that many more details must be specified before implementation is possible. Exactly so. The description suffices for people, but not for machines.

6.2 Differences Between Algorithms and Turing Machines

Algorithms, under Definition 2, are not Turing machines for at least a couple of immediate reasons. We embrace algorithms that are prima facie non-digital, such as Bisecting an Angle and IRS tax instructions. As to whether these could be computerized while retaining their algorithmic identity, the current work takes no position. Furthermore, the last criterion *Provisions and Purpose* is not explicated in terms of Turing machines.

The use of arbitrary values, as in the description of the algorithm Bisecting an Angle, also renders algorithms different from Turing machines. While the invocation of an arbitrary value is beyond the capacity of a computer, I claim that it does not exceed the capacity of the human who sets out to execute an algorithm. Computers can generate specific random values, of course—even though, because generated by a program, they are not entirely random. But a call for an *arbitrary* value—some specific value; it does not matter what—cannot be understood by a machine, but can be understood by a person, without being instantiated.

6.3 Considering Set-Theoretic Correspondences

These differences notwithstanding, however, it would be instructive to probe deeper into the Turing machine analogy and its predication on set-theoretic correspondences. To do so, we can revert to the preliminary Definition 1, free of the *Provisions and Purpose*, and assume that the imperativity requirement is met, somehow, perhaps by expanding the definition of Turing machine to include its operational wrapping.

We proceed under the caution that we are attempting to associate a non-formal object with a formal object. It is unknown, under my definition, whether algorithms form a denumerable set at all (nor have I asked whether algorithms form an infinite set, or even a set). Since this work carefully avoids specification of a symbolic language, and thereby precludes formal construction, it cannot provide a recursive definition of all possible algorithms. But let us consider the possible correspondences, for those who wish to entertain the possibility.

6.3.1 Questions from the Algorithm Point of View

From the algorithms we know by name, along with others that claim a robust conceptual existence, let us take an algorithm and ask: Where is, or are, its Turing machine(s)? The set of Turing machines is vast. It comprises not only those that

"cleanly" implement our algorithm, perhaps Binary Search, in a way that would get the nod from a decent programmer, but also those that pad the sequence of steps with irrelevant switches of values among variables, loop iterations that have no effect, embedded computations that are not used, and extraneous inputs and outputs. Many Turing machines perform (produce the results of) Binary Search in both tidy and messy implementations (Some tidy implementations will maintain the midpoint value at the beginning of the tape and some, elsewhere). Could each of them be performing a unique algorithm—BinSearch1, BinSearch2, etc.? Perhaps, but we have decided that algorithms are minimal by nature, and right now, we are looking for one, namely, Binary Search. So, for two reasons, different "clean" versions and dubious "dirty" versions, there is no one-to-one correspondence between algorithms, construed in our intuitive way, and Turing machines, should anyone propound that there is.

A more likely suggestion is that a given algorithm, like Binary Search, is a set of Turing machines, an equivalence class, based on... what? There is no independent criterion, no effective method, for identifying the members of that class. As we know, we cannot figure out, in finite time, what an arbitrary Turing machine does, based on inspection of inputs and outputs. Under the informal point of view adopted here, which would demand that each member of such a class be checked against the human conception of Binary Search, I venture that we cannot figure out, in human time, what an arbitrary Turing machine does, based on poking around in the symbols and tuples that comprise it. So, given an algorithm, we cannot find the set of Turing machines that implement it, which undermines any constructive claim there is a one-to-many correspondence between algorithms and Turing machines. Furthermore, a Turing machine may perform several algorithms. Since there is a Turing machine that performs every possible program (sensible or not), there are Turing machines that perform Binary Search followed immediately by Insertion Sort followed by Rot13 followed by Boyer-Moore String Search, all on the same or perhaps different sets of variables. Since equivalence classes must be disjoint, an algorithm cannot be defined as "all the Turing machines that perform *this* set of instructions."

Ultimately, paraphrasing the introductory Section 2.2, to say that Binary Search is a class of Turing machines performing a certain procedure that cannot be produced or circumscribed is likely to leave the interested public less, rather than more, enlightened. What the interested public wants to know is—What kind of thing is this *procedure*? What does Binary Search, and its fellows, do? We can answer that. And when we answer it, we do not mention Turing machines.

Suppose that the correspondence between algorithms and Turing machines consists in this: A programmer, given an algorithm A, writes a Turing machine T for it. If this could be done for any algorithm, and we somehow incorporate (or set aside) the preconditions and goals, and provide pseudo-random constants for arbitrary values, we would have an injective (but not surjective, as we reject a "pointless" T) mapping $A \longrightarrow T$. Recall that our definition allows non-digital algorithms, algorithms that do not require a computer under their standard descriptions, but that someone might suggest that they *could be* carried out on a computer without loss of algorithmic identity. If so, then, yes, under these conditions, any algorithm can be implemented by some Turing machine. This, however, is not news, and flirts with begging the question,

as we seem to be forcibly narrowing the scope of "algorithmic" to "Turing-computable."

6.3.2 Questions from the Turing Machine Point of View

Now to approach from the other direction, given a random Turing machine, let us ask: What is its algorithm? Do all Turing machines implement an algorithm (at least one)? What about no-op (that is, do-nothing) and trivial programs—do they implement algorithms? Our Definition 2 requires that a task be accomplished. We will admit those that perform some task of interest to some person, which leaves a broad field. Certainly, some Turing machines do indeed clearly implement algorithms. And some Turing machines generate a string of n zeroes. Some Turing machines generate a single zero. Are these algorithms? Is there an algorithm for looping forever? We could duck the last question on the grounds that "loop" by itself assumes a computer-like setting. But what about the question whether every conceivable computation implements an algorithm, even those that simply output a constant? These trivial tasks do not rise to the level of the algorithms that we name and value. But, for such a case, we could tell a story in which such a task is the goal desired, with the result that all the points of our definition are met.

It would be difficult, however, to countenance algorithms that assign a series of values to variables that are meaningless because they are simply overridden by later assignments, or algorithms that read a value, manipulate it, and then write out the original value, or nothing at all. We recognize an algorithm by what it does, not just by what it produces. To call such self-defeating or inconsequential procedures algorithms violates the "useful" connotation of the word "effective" and violates the meaning of "accomplishing a task." It does not appear that every Turing machine embodies an algorithm.

What about the question of the no-op, an empty or "null" algorithm (corresponding to the no-op Turing machine that does nothing)? To claim that the no-op is a compound control structure that accomplishes a given purpose under given provisions—to claim that its content brings about a change—strains the concept past the breaking point. Therefore, there is no such thing. In the view under development, there is no theoretical need for a null algorithm, or any kind of base case that could serve as a fixed point for an inductive definition. We are not undertaking any such formal definition, and, as a consequence, we can reject any artificial reification. When we claim that, pursuant to the definition, there is no null algorithm, we do so in safety.

What about programs, those tangible written artifacts?—do they all implement algorithms? It depends. Do all programs implement finite, abstract, effective, compound control structures? Yes. Do all programs manifest the abstract, effective, control structure in an imperative form? Maybe; we would have to discuss whether deliberately non-imperative languages are actually rarefied translations of imperative constructs akin to assembly-language operations. To respect the augmented Definition 2, we would have to ask: Do all programs implement the accomplishment of a given task under given conditions? Maybe; we would have to discuss such issues as how these aspects are given—whether inherent in the program itself, or dependent on human attention.

We do not wish to thwart all attempts to associate algorithms with Turing machines, but rather to reveal nuances that should be considered when focusing on the first referent of the Church-Turing thesis, and indeed, to honor that notion, of effective calculability, in its manifestations in the algorithmic objects themselves, and to follow where it leads. This work is not an exegesis of the Church-Turing thesis in any of its variants, but poses little threat to that thesis except in the interpretation of it as a bald claim that an algorithm *is* a Turing machine.

7 Open Questions

Many question arise, among these the issue where to draw the boundaries of the conceptual object *algorithm*, and the connection between human understanding of algorithms and machine implementations. Some paths for future work include the pursuit of a formalization of the imperative, and mining the study done in planning and goal-directed artificial intelligence to elucidate the purposefulness of an algorithm. Practical issues include investigation of identity conditions and distance metrics that would apply to algorithms. Whether specification and computational mechanisms hold a firm position in this picture remains to be seen; that question, and others relating this topic to current discussions in the philosophy of computer science, are open. The work in specification, for example, places it first in the design process, exercising normative influence, but this paper suggests that the algorithm exists, in some way, antecedent to its invocation by a specification. Is that a conflict? If so, perhaps the resolution is simply that a specification may call for an algorithm; or perhaps the resolution is some more complicated and interesting interaction between creation and discovery.

Philosophical considerations invite us to develop the epistemology of the algorithm, especially in the learning of an algorithm as distinct from other subjects, and perhaps deploy experimental philosophy to determine what views of algorithms are held by people, both inside and outside computer science. We can continue to pursue ontological questions into some fairly odd territory. Are there algorithms that are opaque to us, such that we cannot tell exactly what they are doing even though we recognize their algorithmic nature? That depends on other perspectives; for the materialist, no; for the idealist, maybe. Should we admit algorithms that no one can even discern? In other words, does the notion outstrip human apprehension? We have no answer. These worries are not unknown in philosophy. Many marginal cases can be put forth, for any ontological claim. We accept the existence of *virtue*, at least as a legitimate philosophical topic, in spite of the obvious difficulty of circumscribing the virtues, let alone *virtue* itself. On the proposed Definition 2, we accept that the concept has a fuzzy boundary, which invites a closer look.

8 Conclusion

We have attempted to answer Moshe Vardi's question, "What is an algorithm?" from the point of view of computer science on the ground. Although the ontology of the

algorithm has not received wide attention beyond its treatment, by computer scientists, as an abstract formalism, the question can be explored from the intuitive or worldly side. The major results of this work are as follows: (1) there is an object, the algorithm, that can be identified and described; (2) an algorithm is not the same as a Turing machine; and (3) the algorithm is a worthy object of philosophical study. Support for (1) is provided by recognition of extensive practice, in naming, discussing, and teaching. Support for (2) is provided by the intentional and imperative qualities and conceptual identity that forestall a simple mapping between algorithms and Turing machines. Support for (3) is provided by the still-open questions of the relationships among the formal and informal realms of computation as well as the sheer presence of the algorithm in public and professional discourse.

A definition was proposed: An *algorithm* is a finite, abstract, effective, compound control structure, imperatively given, accomplishing a given purpose under given provisions (Definition 2). The definition, examined through the lenses of computer science and of philosophy, is justified by practice and by theory, although, in opposition to traditional assumptions, recursive definitions are ruled out by the imperativity aspect.

Acknowledgments I wish to thank William J. Rapaport of the University at Buffalo for his cogent suggestions and steady encouragement, and to note that any failing in the ideas or treatment is mine alone. Anonymous reviewers improved this work via many good suggestions, and have my sincere gratitude.

References

Cleland, C.E. (2001). Recipes, algorithms, programs. *Minds and Machines*, *11*, 219–237.

Dodig-Crnkovic, G. (2011). Significance of models of computation, from turing model to natural computation. *Minds and Machines*, *21*, 301–322. doi:10.1007/s11023-011-9235-1.

GameTable Online Inc. (2013). Rules: Checkers. Available online at https://www.gametableonline.com/pop_rules.php?gid=20.

Gardner, M. (1970). Mathematical games—the fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, *223*, 120–123.

Gurevich, Y. (2000). Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic (TOCL)*, *1*(1), 77–111.

Gurevich, Y. (2011). What is an algorithm? Available online at http://research.microsoft.com/en-us/um/people/gurevich/Opera/209.pdf.

Hill, R.K. (2013). What an algorithm is, and is not. *Communications of the ACM*, *56*(6), 8–9. doi:10.1145/2461256.2461260. ISSN 0001-0782.

Hopcroft, J.E., & Ullman, J.D. (1979). *Introduction to automata theory, languages, and computation*. Philippines: Addison-Wesley Publishing Co. ISBN 0-201-02988-X.

Kleene, S.C. (1967). *Mathematical logic*: Wiley. ISBN 0-471-49033-4.

Knuth, D.E. (1997). *The art of computer programming, volume 1 (3rd Ed.): fundamental algorithms*. Redwood City: Addison Wesley Longman Publishing Co. Inc. ISBN 0-201-89683-4.

Kreisel, G. (1967). Informal rigor and completeness proofs. In Lakatos, I. (Ed.) *Problems in the philosophy of mathematics*, (Vol. 1 pp. 138–186): North-Holland Publishing Company.

MacCormick, J. (2012). *Nine algorithms that changed the world*. Princeton: Princeton University Press.

Markov, A.A., & Nagorny, N.M. (1988). *The theory of algorithms. Mathematics and its applications*. Netherlands: Springer. ISBN 9789027727732.

Minsky, M.L. (1967). *Computation: finite and infinite machines*. Upper Saddle River: Prentice-Hall, Inc. ISBN 0-13-165563-9.

Moschovakis, Y.N. (2002). On founding the theory of algorithms. Available online at http://www.math.ucla.edu/ynm/papers/foundalg.pdf.

Piccinini, G. (2007). Computing mechanisms. *Philosophy of Science*, *74*(4), 501–526.

Rapaport, W.J. (2012). Semiotic systems, computers, and the mind: how cognition could be computing. *International Journal of Signs and Semiotic Systems*, *2*(1), 32–71.

Rosser, B. (1939). An informal exposition of proofs of Gödel's theorems and Church's theorem. *Journal of Symbolic Logic*, *4*, 53–60, 6. doi:10.2307/2269059. ISSN 1943-5886. http://journals.cambridge.org/article_S0022481200035349.

Sieg, W. (2008). Church without dogma. In Benedikt Lwe Cooper, S.B., & Sorbi, A. (Eds.) *New computational paradigms: changing conceptions of what is computable*. New York; London: Springer.

Turner, R. (2011). Specification. *Minds and Machines*, *21*(2), 135–152. doi:10.1007/s11023-011-9239-x. ISSN 0924-6495.

Turner, R. (2013). The philosophy of computer science. In Zalta, E.N. (Ed.) *The Stanford encyclopedia of philosophy. Fall 2013 edition*.

Uspensky, V.A., & Semenov, A.L. (1981). What are the gains of the theory of algorithms. In Ershov, A.P., & Knuth, D.E. (Eds.) *Algorithms in modern mathematics and computer science, volume 122 of lecture notes in computer science*. ISBN 978-3-540-11157-3 (pp. 100–234). Berlin Heidelberg: Springer, DOI doi:10.1007/3-540-11157-3_27, (to appear in print).

Vardi, M. (2012). What is an algorithm? *Communications of the ACM*, *55*(3). doi:10.1145/2093548.2093549.

Weizenbaum, J. (1966). Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, *9*(1), 36–45. doi:10.1145/365153.365168. ISSN 0001-0782.

Wikipedia (2013). Eliza—Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/ELIZA. Accessed 28 Oct 2013.