Recipes, Algorithms, and Programs

CAROL E. CLELAND
Department of Philosophy & Institute of Cognitive Science, University of Colorado, Boulder, CO, USA

Abstract. In the technical literature of computer science, the concept of an effective procedure is closely associated with the notion of an instruction that precisely specifies an action. Turing machine instructions are held up as providing paragons of instructions that "precisely describe" or "well define" the actions they prescribe. Numerical algorithms and computer programs are judged effective just insofar as they are thought to be translatable into Turing machine programs. Nontechnical procedures (e.g., recipes, methods) are summarily dismissed as ineffective on the grounds that their instructions lack the requisite precision. But despite the pivotal role played by the notion of a precisely specified instruction in classifying procedures as effective and ineffective, little attention has been paid to the manner in which instructions "precisely specify" the actions they prescribe. It is the purpose of this paper to remedy this defect. The results are startling. The reputed exemplary precision of Turing machine instructions turns out to be a myth. Indeed, the most precise specifications of action are provided not by the procedures of theoretical computer science and mathematics (algorithms) but rather by the nontechnical procedures of everyday life. I close with a discussion of some of the rumifications of these conclusions for understanding and designing concrete computers and their programming languages.

Key words: action, algorithm, computer program, effective procedure, precisely specified instruction, quotidian procedure, symbol, Turing machine

1. Introduction

From a preanalytic, intuitive standpoint, a procedure is effective if correctly following it reliably yields a definite outcome. The Euclidean algorithm, for example, is ordinarily thought to be effective (for computing the greatest common divisor of two integers) because correctly applying it to two integers invariably yields their greatest common divisor. Similarly, instructions for rewiring an electrical outlet may be said to be effective in virtue of the fact that correctly following them reliably produces a functioning wall socket. Not all procedures terminate when followed, however. The Wallis' algorithm for computing π provides a salient example. It may be used to calculate successive and exact values of the decimal expansion of π, but it doesn't specify a final step terminating in the full decimal expansion of π. In light of cases like this, the concept of effectiveness has come to be defined in the technical literature in terms of the outcomes of applying individual instructions as opposed to the outcome of completing the procedure as a whole. As Marvin Minsky explains in his classic textbook on computer science (Minsky, 1967, p. 105) "our concern here is not with the question of whether a process terminates with a correct answer, or even ever stops."

Minds and Machines **11:** 219–237, 2001.

Minsky doesn't stop, however, with correctly noting that our concept of effectiveness needs to include non-terminating procedures. He further stipulates (p. 105) that the effectiveness of a procedure be construed in terms of "whether the next step is always clearly determined in advance." In other words, the effectiveness of a procedure depends on how well its instructions *specify* the actions they prescribe. This amounts to a linguistic proposal for analyzing effectiveness, and, indeed, Minsky subsequently characterizes (p. 106) effective procedures in linguistic terms, explicitly referring to them as "well defined" or "precisely described". It is important to keep in mind that Minsky does not view himself as introducing a new proposal for understanding the concept of an effective procedure; he sees himself as explicating the received view among computer scientists.

In previous work (Cleland, 1993, pp. 291–293, 1995, pp. 12–13), I argued that Minsky's account of effectiveness is inadequate and needs to be brought into closer alignment with the intuitive notion of effectiveness. For procedures are said to be effective in virtue of the outcomes they yield, and these outcomes are not themselves specified by the instructions of procedures; they depend upon the (logical or mathematical or conventional or causal) consequences of performing the prescribed actions. The solution I proposed was to add a condition stipulating that (in addition to being precisely specified) the actions prescribed by a procedure reliably yield definite (abstract or physical) outcomes when performed. I never questioned the oft-repeated claim that Turing machine instructions provide us with paragons of precise specification of action, and I defended nontechnical procedures (such as recipes) against the charge of failing to provide requisitely precise specifications of action by appealing to the concept of an idealized follower.

In this paper, I focus on the concept of a precisely specified action, and reach the disconcerting conclusion that the much-lauded precision of Turing machine instructions is a myth. Moreover, an analysis of the instructions of the nontechnical procedures of everyday life–what I shall call *quotidian* (a.k.a. mundane[1]) procedures–reveals that although they are inherently vague, they provide us with more precise specifications of action than either Turing machines or numerical algorithms. In other words, quotidian procedures, as opposed to Turing machines, provide us with the real paragons of precisely specified action! Indeed, as I shall argue, a careful look at the prescriptions of action provided by Turing machines reveals that they don't even provide us with *bona fide* procedures, let alone effective procedures; at best, they may be said to provide us with procedure schemas. I close with a discussion of some of the ramifications of my conclusions for interpretation and design of concrete computers and their programming languages.

2. Recipes (and Other Quotidian Procedures)

We shall begin by investigating the precision of quotidian procedures since they are often rejected as ineffective solely on the the grounds that their instructions fail to provide requisitely precise specifications of action; thus the fact that correctly

following a recipe invariably results in a particular outcome is judged not to be sufficient for concluding that the recipe constitutes an effective procedure. First, however, we need to get some terminology and distinctions out of the way.[2] The following recipe will serve as an example.

> Pat the mozzarella balls dry with absorbent kitchen paper. Cut into slices with a sharp knife and arrange on a serving platter with the tomatoes and basil leaves. Squirt on a little lemon juice, season with salt and black pepper, and drizzle with olive oil. Toss gently and serve immediately (Biuso, 1997, p. 15).

The recipe consists of instructions. Each instruction-expression (e.g., "pat the mozzarella balls dry with absorbent kitchen paper") makes reference to an occurrence which is to be brought about (done) as opposed to undergone or merely happen. That is, each instruction designates an action, more specifically, since different chefs may apply the same instruction, an action-type (vs. token). The instructions are expressed by imperatives (e.g., "cut into slices . . . "), indicating that the follower is to perform the designated action-types. In other words, 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 otherreferential relations.

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. The same recipe may be expressed in many different ways, including in different languages and in different terminology in the same language. Also, some quotidian procedures (e.g., for tying shoelaces) may be communicated non-verbally. Different instruction-expressions express the same instruction only if they prescribe the same type of action. Accordingly, the identity of a quotidian procedure depends upon the identity of the action-types prescribed by its instruction-expressions. The order in which the instructions are to be carried out is also crucial to the identity of a quotidian procedure. Failure to perform the prescribed action-types in the specified order in time constitutes just as much a failure to follow the procedure as not performing them at all. In our recipe, the order is specified by the order in which the instruction-expressions appear in the recipe, e.g., the instruction to pat the mozzarella balls dry comes before the instruction to slice them, and this is the order in which the specified actions are supposed to be performed.

We are now ready to investigate the manner in which quotidian procedures specify the actions they prescribe. Human actions are typically designated in terms of their consequences. These consequences are usually causal or conventional, as opposed to logical. As an example, consider an action of raising one’s hand. The concept of a raising of a hand includes the concept of a rising of a hand; in the absence of the latter there can be no occurrence of the former, and any competent speaker of English understands this. Similarly, the concept of signaling includes the concept of someone being signaled. A raising of a hand will not count as a

signaling if no one interprets it as a signaling; at best, it may be characterized as a failed attempt at signaling. It is important to keep in mind, however, that an action cannot be identified with its causal or conventional consequences. Not all risings of hands are raisings of hands. A rising of a hand might, for example, be produced by a random twitch of some muscle fibers, and hence fail to qualify as something that was done (an action) as opposed to something which merely happened or was undergone. Similarly, a raising of a hand may be incorrectly interpreted as a signaling.

The instructions of quotidian procedures exploit the conceptual connection between an action and its consequences. The instruction-expression "slice the cheese", for example, prescribes that a certain state of the world (sliced cheese) be brought about. But it doesn't specify how to do it, whether to use your right hand or left hand, how to hold the knife against the cheese, etc. It is assumed that a follower of the recipe already knows how to bring it about. Some quotidian instructions specify the actions they prescribe in terms of dynamic events, e.g., "kick the ball", as opposed to states. In bringing about a kicking of the ball, the follower performs an action of the required sort. But to recapitulate, the instruction expression does not specify how to kick the ball and the mere fact that an event of a kicking of a ball occurs is not sufficient for there to have been an action of kicking the ball; I might have accidentally struck it with my foot while running across a field. Finally, some quotidian instructions prescribe temporally extended activities that initiate, sustain, and/or modulate continuous physical processes (as opposed to prescribing transitory actions producing states or fairly delimited events). As an example, consider the following instruction from the Boy Scout method for starting a fire: "turn the spindle with long, steady strokes of the bow;" In such cases, it is often difficult to distinguish the process from the activity that initiates, sustains and/or modulates it. For the activity and the process overlap in time and are intimately interconnected. Nevertheless, as in the case of a raising of a hand and a rising of a hand, there is a difference between an action of turning a spindle and a turning of a spindle; the presence of the latter is possible in the absence of the former. In short, although they are intimately connected with their consequences, actions (whether transitory or extended) are nevertheless distinct from them.

Specifying actions in terms of their causal consequences (whether states, events, or processes) has the advantage of providing fairly precise criteria for the satisfaction of an instruction. To the extent that we can recognize sliced cheese, kicked balls, turning spindles, etc., we can determine that we have correctly applied an instruction. On the other hand, it also means that it is possible for someone to recognize when an instruction has been correctly applied without knowing how to apply it. The judges in gymnastics and figure skating competitions provide good examples of this phenomenon; although they can recognize when various actions have been correctly performed, few of them know how to perform them. Viewed from this perspective, it would be better if quotidian instructions specified how

to perform the actions they prescribe, as opposed to merely providing us with characterizations of their consequences.

Unfortunately, this can't be done. For although complex actions may be analyzed in terms of simpler actions, we eventually reach basic bodily actions.[3] Basic bodily actions are specified in terms of their direct effects, which are basic bodily motions. Like non-basic actions, basic actions are distinct from their consequences; there is a difference between a movement of my finger and my moving my finger. This brings us to the question of what distinguishes behavior that is a consequence of action from behavior that is not. Most contemporary philosophers agree that it involves being preceded in the right way by a special kind of psychological state, namely, an intentional (content bearing) mental state. But this is where agreement ends. There is little consensus about the identity of the intentional mental state and the nature of its connection to behavior. Moreover, all the proposed accounts of the connection (and they run the gamut from causal to non-causal) face serious problems. Fortunately, I don't need solutions to these difficult problems. All I need is a minimal intuitive concept of action. For however action is ultimately analyzed, it will still be true that actions are things that are done, as opposed to undergone or merely happen, and, most importantly for my purposes, that they are specified in terms of what is to be brought about (their non-logical consequences), as opposed to how to bring it about.

3. Algorithms

So how do the more refined procedures of mathematics compare to quotidian procedures? We will focus on numerical algorithms since they are the prototypical examples from mathematics. Consider the following version (Hennie, 1997, p. 9) of the Euclidean algorithm for computing the greatest common divisor of two positive integers:

Step 1: Let b and a be specific positive integers. Compute the remainder of a with respect to b. Call this remainder r, and proceed to Step 2.

Step 2: If $r = 0$, terminate the computation with b as the result. If $r \neq 0$, replace a by b and b by r. Now repeat Step 1.

At first glance, it seems to conform to my analysis of the precision of specification provided by quotidian procedures, the obvious difference being that many of the action types prescribed are numerical. A little reflection on the nature of the specifications provided by its instructions, however, reveals some significant differences.

Unlike quotidian instructions, most numerical instructions do not designate the action-types they prescribe in terms of their consequences, which are numerical values as opposed to states, events, or processes. This is especially obvious in the case of operations involving large numbers. The concept of dividing 4,724 by 1,181, for example, does not include the concept of 4 like the concept of raising a hand includes the concept of a rising of a hand or the concept of turning a spindle

includes the concept of a turning of a spindle. But this is just as true of numerical operations involving small numbers. For as Kant pointed out some time ago (Smith, 1965, pp. 52–53), considered just in itself, the instruction ‘add 5 to 7’ does not "include" the concept of 12 in its meaning. It is important to appreciate that appealing to past experience cannot circumvent this problem. Associating 12 through experience with adding 5 to 7 is not the same as the concept of 12 being included in the meaning of the instruction ‘add 5 to 7’. This is not to deny that Kant’s notion of "meaning inclusion" is vague and metaphorical. The distinction I am drawing, however, is not quite the same as the much maligned analytic-synthetic distinction. For the conceptual connection between raisings of hands and *risings* of hands and *turnings* of spindles and *turning* spindles is much closer and more obvious than the conceptual connection between *bachelorhood* and *maleness* or *triangularity* and *three sidedness*, being reflected in the very terminology used to designate actions and their consequences. Numerical instructions do not exhibit any obvious conceptual connection between the numerical operations they prescribe and the numerical consequences of performing them, and this, I submit, makes them significantly different from quotidian instructions.

If algorithms don’t specify numerical operations in terms of their numerical consequences, then, surely, they must specify them in terms of the activity required to produce those consequences; otherwise, how can a numerical instruction be said to tell one what to do? Unfortunately, this does not seem to be the case either. The objects ostensibly manipulated (numbers) are neither perceptible nor causally efficacious. Besides, even if we identify numbers with numerals, numerical instructions still can’t be said to provide us with specifications of action. For although action presupposes things to manipulate, having something to manipulate isn’t enough to constitute action. In our recipe, for instance, it wouldn’t have sufficed to specify cheese and a knife. One also had to specify what is to be done to the cheese by the knife, namely, slice it (as opposed to stab it, beat it, dice it, etc.). But it isn’t at all clear that we manipulate numerals qua physical objects (inscriptions) when we satisfy numerical instructions. In the first place, we often perform numerical operations in our heads; in contrast, we can’t slice cheese in our heads. Furthermore, in contemporary mathematics the idea that numerical operations involve genuine manipulations has been eliminated. Numerical operations are identified with functions which, in turn, are identified with sets of ordered pairs (whose elements may be numbers or, themselves, ordered pairs of numbers), and sets of ordered pairs are completed structures, as opposed to ways of doing things. In short, we really don’t know what we are doing when we perform a numerical operation, and if we don’t know what is involved in performing a numerical operation, we can hardly describe how to do it.

One cannot circumvent this problem by maintaining that our knowledge of how to perform basic numerical operations is grounded in previous experience. For the fact that we have successfully performed a numerical operation isn’t sufficient to fix its application in a new case. Kripke’s famous numerical question (Kripke, 1982,

pp. 20–21) provides a good example. Assuming that I have never before performed addition on numbers greater than 56, what determines that ‘125’ is the correct answer to the question "68 + 57= ?"? Nothing that I have done previously can determine that "+" means addition (where addition is identified with a unique set of ordered pairs). For there are infinitely many sets of ordered pairs which are just as compatible with the finite number of examples of addition to which I have been exposed up until the time I apply the instruction, including "quus", which I symbolize by "$*$" and Kripke defines as follows: $x * y = x + y$ if $x, y < 57$, and $x + y = 5$ otherwise. In other words, everything I have done up to the point of supplying an answer to the question at hand is as compatible with the answer ‘5’ as it is with the answer ‘125’. Indeed, everything I have done is consistent with every possible answer since to each possible answer there corresponds a function compatible with my earlier experiences with numbers less than 57. It should be obvious that this difficulty cannot be resolved by supplying a more precise description of "+". For what, short of stating in full the infinite set of ordered pairs defining the function of addition, could possibly pin down that I mean addition by "+" and not some other function. Kripke concludes that nothing other than the brute contingent fact of widespread agreement fixes the correct application of a numerical instruction.

Kripke extends his conclusion about numerical instructions to instructions in general, contending that for any instruction whatsoever there are always an indefinite number of possible hypotheses about its future use which are consistent with its past use. His position amounts to a sweeping inductive skepticism about meaning. Goodman’s new riddle of induction (which shows that past experience is insufficient to determine that, for example, "green" means green as opposed to grue[4]) supplies the classic argument for non-numerical expressions (see Goodman, 1983, pp. 72–83). It is important to keep in mind, however, that Kripke’s skeptical argument is based upon the assumption that the only possibility for fixing the meaning of an expression is its past use. If this dubious positivist assumption is false (and it does seem natural to construe his argument as a *reductio*!) the skeptical conclusion about meaning doesn’t go through. But we are still faced with the problem of making sense of how numerical instructions can be said to provide precise specifications of the operations they prescribe. For to recapitulate, we don’t know what is involved in performing a numerical operation and numerical instructions don’t provide us with criteria for their successful application. This leaves us in a much worse predicament than we are in the case of quotidian instructions since (eschewing extreme inductive skepticism about meaning) quotidian instructions provide us with criteria for their successful application; although they don’t tell us how to perform the actions they prescribe, they do specify recognizable consequences of performing them.

I am not denying that we correctly perform numerical operations in response to numerical instructions. This is not the issue with which we are concerned. We are concerned with the question of how "precisely" an instruction may be said to specify the action-type it prescribes. And the disturbing conclusion to which we are

being led is that quotidian instructions provide us with more precise specifications of action than do numerical instructions. It is important to keep in mind that the concept of precision with which we are concerned is highly specialized. It has little to do with the idea of eliminating vagueness or increasing numerical accuracy.[5] No one denies that quotidian instructions are vague in the sense that there will always be cases where we are genuinely uncertain as to whether a given instruction has been successfully executed. A good example is a small child who, in response to the instruction "slice the cheese", cuts cheese into large, irregular chunks. The instruction followed by the child can be made less vague by rewording it and specifying that the widths of the slices not exceed a certain value and that the slices conform to more specific geometrical constraints. Numerical instructions do not lack precision in this sense. Furthermore, while it is true that, before completing a prescribed computation, we can often "guess" the range of numbers within which the solution falls, this is not because the instruction contains reference to the range of numbers concerned. It is because we have had previous experience with numbers falling within that range or have already completed a partial computation of the answer. The meaning of the instruction ‘add 4,323 to 2,562’ does not include the concept of the range of numbers between 6,500 and 7,000 any more than it includes the concept of 6,885. In other words, numerical instructions do not provide numerically inaccurate specifications of their consequences either. Numerical instructions lack precision in Minsky’s sense, namely, we cannot make good sense of the claim that they determine in advance what a follower is to do, and no amount of fiddling with their wording can improve the situation. In contrast, although they do not tell us how to perform the actions they prescribe, quotidian instructions do provide us in advance with criteria for their successful application, criteria which, while they can never be completely freed from problems with vagueness, may be refined and made more definite. Thus there is a sense in which they can be said to (albeit inexactly) determine in advance what a follower is to do. It is in this sense–the sense crucial to the contemporary concept of an effective procedure–that the instructions of quotidian procedures may be said to provide us with more precise specifications of action than those of numerical algorithms.

4. Turing Machines and Their Programs

The standard retort to concerns about the precision of specifications of action provided by numerical algorithms is to point out that they may be reformulated as Turing machine programs. Turing machine programs are supposed to provide us with procedures whose instructions are paragons of precisely specified action; unlike quotidian procedures, they are supposed to provide us with perfect "precision" in Minsky’s sense of the word, i.e., to unequivocally determine in advance the next step.

Although they are sometimes given formal mathematical characterizations, Turing machines are most commonly characterized informally as very general, abstract

mechanisms. Let us therefore begin our discussion of Turing machines with their informal characterization. A Turing machine is said to consist of a "mechanism," known as a "finite state machine," coupled to an external storage medium called the "tape". The tape, which could be represented by anything (from a bunch of tin cans to graph paper) is divided into squares. At any given moment, the tape has only finitely many squares. However, an indefinite number of additional squares may be added to either end, making it effectively infinite. Each square of the tape is occupied by at most one of a finite number of distinct symbols; the set of symbols, usually represented by {SO, ... Sn}, is called "the alphabet" of the machine. The finite state machine is linked to the tape through a "head" which is said to be "positioned" over one of the squares. At any given moment, the machine is characterized as being in one of a finite number of "internal states", qi, . . . , qm. As is the case with a concrete machine, these states completely define the Turing machine's mechanism, i.e., the finite state machine. However, unlike the internal states of a concrete machine, the internal states of a Turing machine are not physical structures. They are explicitly identified with instructions. A Turing machine is said to be in a specific state qi only if it is about to carry out a specific instruction i. It doesn't matter how qi is physically realized. All that matters is that the machine is about to carry out instruction i.

Turing machines are said to "follow" procedures called "programs." A Turing machine program is extensionally defined by its instruction set. Unlike the instructions of quotidian procedures and numerical algorithms, the instructions of Turing machine programs are always conditional in form; an initial configuration of the machine is necessary (and always provided) to fix the sequence of actions. The instructions prescribe that the head do certain very simple things (such as move to the right one square) depending upon what symbol is in the square of the tape currently being scanned. There are a number of different ways to represent a Turing machine program, including quadruples, flow graphs, and machine tables. The following machine table provides a good example of a Turing machine program:

	S0	S1
q1	S1q1	Lq2
q2	S1q2	Lq3
q3	S1q3	Slq3

The first row of the table represents a single instruction which prescribes two possible outcomes depending upon what symbol is being scanned by the head, viz., write an S1 and go into state q1 if scanning an SO, or move the head to the left one square and go into state q2 if scanning an S1. One can see how the machine operates by inspecting the following representation of its moment by moment configurations:

0000**0**00	0000**1**00	000**0**100	000**1**100	00**0**1100	00**1**1100
q1	q1	q2	q2	q3	q3

Each sequence of numbers represents the contents of the machine's tape during one of its configurations; the sequence to the far left represents the initial configuration (each square contains an SO, represented by a "0", and the machine is in state q1) and the sequence to, the far right represents the final configuration (three consecutive S1' s, represented by "111", on an otherwise "blank" tape). The q1, which appear below each sequence, represent the state of the machine during that particular configuration; each of the qi are positioned directly below the symbol (in bold type) which is being scanned by the head at the moment in question. Thus one can see that the machine table represents a Turing machine program that prints three consecutive S1's when started in state qi on a tape filled with S0' s.

We are now in a position to explicitly compare Turing machine programs to quotidian procedures for the purpose of assessing the precision with which they specify the action-types they prescribe. The instructions of Turing machine programs are expressed in the form of common English imperatives (e.g., *write* an S1, *move* to the left one square), and hence seem to prescribe actions in the same sense that quotidian instructions prescribe actions. Moreover, the order in which these action-like occurrences are performed is predetermined in time; it is secured by the complex conditional nature of the instructions and the initial configuration of the machine. A closer look, however, suggests that the ostensible similarities between Turing machine programs and quotidian procedures are misleadingly metaphorical.

Let us begin with Turing machine symbols since these are the entities ostensibly manipulated ("written" and "erased") by Turing machines. Turing machine symbols are traditionally characterized as purely formal. They are said to be purely formal in the sense that the ways in which they are manipulated depend only upon their "shape" (vs. content). Thus the shapes of Turing machine symbols are crucial to the identity of the action-types prescribed by Turing machine instructions. The question is can we make good sense of the claim that Turing machine symbols have shape?

Considered just in itself, independently of its physical realizations, a Turing machine is an abstract machine-type. One might therefore think that the shape of a Turing machine symbol could be determined by generalizing across different physical realizations of the same Turing machine. The idea is to identify the shape of a symbol with whatever geometrical feature is had in common by all of its tokens. But this won't work. Across different physical realizations of the same Turing machine, the same symbol may be instantiated by objects having incommensurable shapes, e.g., pencil squiggles, punched squares, pebbles in tin cans. Moreover, we needn't use geometrical features to encode Turing machine symbols. We could use weights or colors, or even properties of events (e.g., the duration of a flash of light). The point is any definite but distinct physical properties may be used to encode and distinguish the symbols of a Turing machine (type). Indeed, the only physical

constraints on tokens of Turing machine symbols hold within (vs. across) physical realizations of Turing machines, namely, all tokens of the same symbol must have in common some (it doesn't matter what) physical property and all tokens of different symbols must differ in some physical property.[6] Thus, although different code-types for Turing machine symbols (which will be specific to particular realizations of Turing machines) may be distinguished in terms of the physical properties of their tokens, different Turing machine symbols (qua constituents of a specific Turing machine considered independently of any of its instantiations) can not be so distinguished. For the most that may be said about Turing machine symbols *per se* is that tokens of different symbols have different but not any definite physical properties. At best this gives us a relation of bare physical difference among symbols. Unfortunately, a relation of bare physical difference isn't enough to provide us with the concept of the shape (however broadly construed) of a Turing machine symbol since it is an external relation among symbols and external relations aren't capable of individuating their relata.

In the context of the formal mathematical analysis, even this minimalist conception of the shape of a Turing machine symbol must be relinquished. Mathematicians analyze Turing machines in terms of mathematical structures. Mathematical structures consist of functions and relations (both of which are identified with sets of *n*-tuples, ordered in the case of the former), and constants. The prototypical ("usual") mathematical structure for a Turing machine includes three binary functions (the "next place" function, the "next symbol" function, and the "next state" function), and two symbols, which are commonly said to be the integers 0 and 1.[7] It is important to appreciate that these integers can't be viewed as numerals (which have shape) since this would amount to conflating an abstract mathematical structure (viz., the usual structure for the Turing machine) with its concrete representations. So it seems that they must be bona fide integers. But integers (qua abstract mathematical objects) have no physical characteristics. This suggests that the only thing that can serve to distinguish Turing machine symbols is bare numerical difference, which completely precludes the possibility of making sense of the claim that they have unique distinguishing structures, however minimal. Finally, mathematicians do not identify Turing machines with their prototypical mathematical structures; they identify them only up to isomorphism. That is to say, considered independently of a particular realization, a specific Turing machine is a class of isomorphic structures (both abstract and concrete). Viewed from this perspective, Turing machine symbols amount to nothing more than logical roles in a second order structure, which, considered in itself, is no more a Turing machine than the set of all red objects is a red object. As such, Turing machine symbols are best thought of as conveniences of discourse, as opposed to symbols of some intangible and rarefied sort. Until we get to the level of the individual structures in the equivalence class defining a Turing machine we don't have genuine symbols. We have placeholders for symbols–symbol variables. As a consequence, we can't

be said to have specifications of action, however imprecise, at the level of a Turing machine considered independently of any of its instantiations.

Furthermore, even supposing that Turing machine symbols were bona fide symbols, the instructions of a Turing machine program still couldn't be said to provide us with precise specifications of action. For as discussed earlier, although action presupposes things to manipulate, having something to manipulate isn't enough to constitute action. Despite the use of familiar English expressions for action such as "erase" and "write", Turing machine instructions do not specify what is to be done once their symbol variables are replaced by symbols. What counts as erasing and writing a symbol is left completely open. If pebbles replaced the variables, for instance, erasing a pebble could be realized by activities as diverse as pulverizing it, painting it, or removing it from a tin can, to mention just a few possibilities. This difficulty is accentuated in the formal mathematical account since there is no requirement that the structures in the equivalence class defining a specific Turing machine even be dynamic. For the basic Turing machine operations (erase, write, move) are defined in terms of ordered *n*-tuples (mathematical functions), which do not require change (let alone change which qualifies as action) for their realization; they may be instantiated by spatially as well as temporally ordered structures. As a mathematician once cheerfully conceded to me, a Turing machine could be realized by a quartz crystal! In other words, insofar as the operations specified by a Turing machine are given a formal mathematical interpretation, they cannot be said to represent actions, however indefinite.

To the extent that they do not provide us with specifications of action, Turing machine instructions can not live up to their promise of providing us with more precise specifications of action than quotidian instructions. Indeed, they cannot even be said to provide us with specifications of procedure since, as I have argued, the concept of action is essential to the concept of procedure. At best, they may be said to provide us with procedure schemas, i.e., temporally ordered frameworks for procedures. When these schemas are filled in with bona fide specifications of action, we get genuine procedures.

5. Concrete Computers and Their Programs

This brings us to concrete computers and their programs. A modern digital computer executing a program is commonly characterized as "being" (realizing or instantiating) a Turing machine. As an example, consider the following program for determining the greatest common divisor of two positive integers. It is written in an artificial languagecalled BASIC.

```
10   INPUT "FIRST NUMBER =", A
20   INPUT "SECOND NUMBER =", B
30   Q=INT(A/B)
40   R=A-B*Q
```

```
50   IF R> 0 THEN GOTO 60 ELSE GOTO 90
60   A=B
70   B=R
80   GOTO 3O
90   PRINT "GCD =", B
```

The BASIC program prescribes what seem to be bona fide actions. Considered just in itself, however, it cannot support the claim that concrete computers executing programs are instantiations of Turing machines. The number of symbols and action-types exceeds those of the usual structure for a Turing machine (which, it will be recalled, is limited to two distinct symbols and three distinct action-types). Put another way, the BASIC language is too rich to underwrite a relation of isomorphism between a computer executing a typical BASIC program and the usual structure for a Turing machine.[8] It should be clear that this is true for all high level languages, e.g., JAVA, LISP, PASCAL, C, Ada.

The BASIC program closely resembles a numerical algorithm. Some of the instructions are couched in terms of common English imperatives (e.g., lines 80 and 90) whereas others are expressed as familiar functions with one unknown value (e.g., lines 30 and 40). Indeed, a BASIC programmer can follow it just as easily as she can follow the Euclidean algorithm presented earlier.[9] Nevertheless, construed as something that can be followed by a human being, the program doesn't represent an improvement over the Euclidean algorithm. The instruction in line 40, for example, can no more fix for a human follower what counts as its correct or incorrect application than the instruction in "Step 1" of the Euclidean algorithm to compute the remainder of *a* with respect to *b*. In neither case does the instruction provide criteria for determining that an action of the required type has been successfully completed. Moreover, like the Euclidean algorithm, the instructions of the program don't specify how to carry out the numerical actions they prescribe; it is merely assumed that a follower knows how to do it. In short, construed as something that may be followed by a human being, the BASIC program represents no improvement over the Euclidean algorithm; it seems to be just a different expression of it.

There is, of course, a striking difference between a computer executing the BASIC program and a person following the Euclidean algorithm. We know how the computer works–how the machine's software and hardware translate the instructions into a sequence of electronic states and how the circuits of the machine subsequently transform these states into other states. As a consequence, one may feel confident that the sequence of electronic states that the machine passes through are causally determined in advance by the program. In contrast, we don't know what a human being does when she follows the Euclidean algorithm, and hence aren't very confident that what she does is similarly determined in advance; our uncertainty is exacerbated by the fact that humans frequently make mistakes when following algorithms. Viewed from this perspective, the BASIC program seems to

provide us with a more precise (in Minsky's sense) method for determining the greatest common divisor of two integers than the Euclidean algorithm.

It is important, however, not to confuse knowing (at least in principle) how a computer executes a line of code with knowing whether it did what it was supposed to do, namely, determine the quotient of two positive integers. This is a subtle point. In one sense a computer may be said to apply an instruction correctly if there are no hardware or software failures while it is processing the line of code; it did just what it was told to do. In another sense, however, the machine may be said to apply the instruction correctly only if it gets the result that a human being would get if she applied the instruction correctly. But in the latter sense the machine is no better off than a human following the same line of code. For the computer may be judged to have successfully executed the line of code but still not gotten the correct result. In other words, even supposing that the program causally determines in advance which physical states the computer passes through, it doesn't follow that what the computer does is to compute the greatest common divisor of two positiveintegers.

The preceding discussion underscores a frequently overlooked point. Whether a computer or a human being determines the correct values of a specific numerical function when following a procedure expressly designed for this purpose is secured neither by the procedure itself nor by what the human or computer does when following it.[10] There may be very good reasons for believing it to be true, reasons based upon well entrenched, theoretical mathematical considerations. The use of inverse functions to check one's calculations when balancing a checkbook provides a familiar example. Identity relations among functions are similarly utilized to test the numerical operators of computers. But while such tests may increase one's confidence in the identity of a function, they provide no guarantees. The claim that a function being computed is division, for instance, is at best an empirical hypothesis, and like all empirical hypotheses open to challenge. Any concrete computer has a finite life span, and hence actually computes only partial functions. Given any partial function, there are an infinite number of different ways of continuing it, and each of these yields a different total function. In addition, there are the inevitable hardware and software failures which afflict all concrete computers, resulting in the computation of partial functions which are, technically speaking, inconsistent with the total functions (e.g.,division) the machines allegedly compute. All of these problems are conveniently glossed over by the misleading metaphor of a Turing machine as an abstract but nevertheless bona fide mechanism. For it makes a concrete computer seem more like a Turing machine than it really is, and insofar as a Turing machine is identified with a formal mathematical structure the conjecture that a concrete computer computes a particular function may seem more like a mathematical theorem than an empirical hypothesis; it may seem that computer science is more mathematics than science.[11]

The manner in which a computer processes a program is quite different from the way in which a human being processes a numerical algorithm. The computer can't be said to "understand" the instruction-expressions except insofar as it has another program (an interpreter) to "translate" the high level BASIC instructions into its machine language; a computer lacking such a program can't apply the instructions. Indeed, one could by-pass the interpreter and code the program directly in machine language as strings of octal (or hexadecimal) numerals. This makes what the computer does seem more like following a Turing machine program; for strings of octal numbers have binary equivalents. Nevertheless, the computer can't be said to understand strings of numerals any more than it can be said to understand BASIC instructions. For there aren't any numerals (let alone numbers) inside the machine; at best, the computer may be said to accept numerals as input and to produce numerals as output (on printout, monitor screens, etc.). What is inside the computer are circuits etched on silicon chips, and what the computer does when it executes a program is to open and close these circuits in causally predetermined ways. This activity is the only thing the computer may be said to "understand". The use of English imperatives, mathematical formulae, and sequences of numerals are solely for the benefit of programmers who are trying to get the machine to exhibit overt behavior like that of a mathematician following the Euclidean algorithm. But what the computer actually does in response to the instructions of the BASIC program resembles neither the psychology nor the neurophysiology of human thought processes; what the computer does is pass into and out of electronic states.

What makes concrete computers such powerful and useful devices is neither their instantiating Turing machines nor their providing us with more precise methods for computing numerical algorithms but, rather, their capacity for emulating a human being following a quotidian procedure. Just as the salad recipe specifies in advance that cheese be sliced by a knife without specifying how this is to be accomplished so the BASIC program causally determines in advance that certain things be achieved (some of which are given numerical interpretations by humans) without determining how they are to be accomplished. Not only can different types of computer (having different physical components and architectures) execute the same BASIC program, but the same computer executing the same BASIC program at different times will (depending upon what else it happens to be doing) activate different circuits and chips. In other words, computer programs underdetermine the behavior of the machines that implement them. This is the source of the power and utility of the modern digital computer. Programmers and users can specify in the context of their interests, training, and purposes what is to be done without having to concern themselves with the messy physical details required to bring it about.

It is important to keep in mind just how different this is from what a Turing machine (qua abstract, logical "device") reputedly does when it "follows" a program. The program logically predetermines *everything* that is done by the machine;

nothing is left open.[12] As a consequence, there isn't a distinction between *what* the machine does and *how* it does it. In contrast, no computer program (considered just in itself) provides a complete specification of the behavior of the machine implementing it; even the lowest level programs (machine language programs) depend upon a complicated encoding scheme called the "machine code format" to fix the physical states the machine actually passes through. This gap between the hardware and the software of a concrete computer mirrors the gap between the specifications provided by a quotidian procedure and the differing ways in which a human being may satisfy them. Just as a quotidian procedure underdetermines the behavior of a human being by leaving open how its specifications are to be realized, so a computer program underdetermines the behavior of a concrete machine by leaving open how its specifications are to be realized. In other words, the power and utility of the modem digital computer resides in its almost human plasticity, and this plasticity is a consequence of our having succeeded in automating the following of a quotidian procedure.

If I am right about the nature of the significant relation between concrete computers and their programs some popular views about the nature of minds and machines need rethinking. What computers executing programs mimic is the overt behavior of humans following quotidian procedures, and the overt behavior of a human following a quotidian procedure does not in an illuminating way resemble the neurophysiological or psychological processes underlying it. Similarly, the physics of a computer processing a program does not resemble the behavior explicitly prescribed by the program. Indeed, the whole point of a quotidian procedure is to isolate and insulate a sequence of overt behavior from the messy causal details of producing it. But if this is so, the celebrated computer metaphor for cognition seems fundamentally misguided; for it is founded on the idea that having a mind is just a matter of executing the right program.

Moreover, my account suggests that it is a mistake for theoretical computer science to focus on Turing machines as paragons for the design of concrete computers. For there is no more reason to require that the physical processes causally generating the input/output functions specified by programs resemble the logical activity of Turing machines than there is to suppose that the neurological or psychological processes of a chef following a recipe resemble her overt activity in the kitchen. Instead, the focus should be on physics, more specifically, on identifying and exploiting physical processes for achieving results that are currently unobtainable or could be obtained in a more rapid or efficient manner.

Indeed, on the account just adumbrated, the central concern about Turing's tantalizing but enigmatic oracles disappear.[13] Oracles are logical black boxes for carrying out uncomputable tasks. They accept digital input and produce digital output, and hence can be integrated into standard Turing machines, allowing them to compute functions they otherwise couldn't compute; Turing called these enhanced Turing machines "O-machines". As an example, consider the infamous "halting problem", which is the problem of designing a Turing machine that computes

the function $f(m,n) = 1$ or 0, where m is a binary number (string of Os and 1s) representing a Turing machine and n is a binarynumber representing the contents of the machine's tape. If machine m halts when started with n on its tape, the value of $f(m,n)$ is 1; otherwise it is 0. It is well known that a universal Turing machine cannot compute this function; it is Turing uncomputable. But one can imagine augmenting a universal Turing machine by an oracle taking the arguments of the function as input and returning the appropriate value as output.[14] The obvious worry about oracles is that their internal operations are a mystery; Turing did not provide them with a positive characterization, remarking only that they work by "unspecified means". On my view, however, this isn't a serious conceptual difficulty since the whole point of a concrete computer is to provide us with the ability to specify tasks independently of the physical details of carrying them out.

Of course, the legitimacy of the abstract concept of an oracle doesn't establish their physical possibility; that is a matter for physics to decide. Nevertheless, it would be a mistake to conclude that oracles are of only theoretical interest because we could never confirm that a proposed physical candidate produced something genuinely Turing unoomputable. For there is no more reason to insist that a physical device couldn't compute a Turing uncomputable function on the grounds that it is impossible to conclusively verify it than there is to insist that my hand calculator can't compute division on the grounds that it is impossible to conclusively verify it. To recapitulate, the claim that any physical entity (machine or human) computes a total function is an empirical hypothesis. Just as there may be good reasons for thinking that my calculator computes division so there might be good reasons for thinking that some mysterious, physically real, black box solves the halting problem. Unfortunately, however, it is beyond the scope of this paper to pursue this issue any further.[15]

Notes

[1]In earlier papers (Cleland, 1993, 1995), I called ordinary, everyday procedures such as recipes and methods "mundane procedure" But after receiving many comments about having given them too mundane a name, I have decided to rename them "quotidian procedures"!

[2]This material was developed in an earlier paper (Cleland, 1993, pp. 287–288).

[3]Some philosophers reject the claim that the basic actions are bodily actions, contending that bodily actions arise out of still more basic actions (variously called "volitions", "willings", or "tryings") which are purely mental. They claim that purely mental actions are needed to explain cases where one unsuccessfully attempts an overt bodily motion, e.g., tries to move a paralyzed limb. But even if this is the case, it doesn'taffect my point here since the actions prescribed by quotidian instructions are bodily actions.

[4]Grue is the "property" of being either green or first examined before t (where t is some specific but wholly future time), or blue and not examined before t

[5]I am grateful to Fred Kroon for pointing this out to me.

[6]I am using the term "property" in what is sometimes called the sparse sense; i.e., I am excluding disjunctive and negative properties.
[7]The usual structure for a Turing machine is most often formulated by mathematicians in terms of a universal Turing machine, but this fact is irrelevant to the point I am making here.
[8]The relation between a high level program and its physical realization in a particular machine is controversial. Many computer scientists hold that a computer program ("software") is an abstract machine made out of text. Others have argued that a program is constructed not out of text but rather physical things such as electron charges and magnetic fields. For a review of this debate, see Colburn (1999, pp. 3–19). Both extremes strike me as confused, and I suspect that this confusion is at the root of the view that a computer executing a high level program (which doesn't look at all like a Turing machine program) is nonetheless an instantiation of a Turing machine program. The confusion arises out of the mistaken view that there isn't a fundamental difference between a procedure and the processes that realize it.
[9]Admittedly, not all high level programs for computing the Euclidean algorithm will as closely resemble the Euclidean algorithm. A good example is a JAVA program. But a JAVA programmer will be able to follow it just as easily as she can follow the numerical algorithm.
[10]For a more detailed discussion of these issues see my "Effective Procedures and Computable Functions" (1995, pp. 17–20).
[11]As James Fetzer has discussed (Fetzer, 1988, pp. 1048–1063, 1991, pp. 197–216), the notion that computer science is a branch of pure mathematics is very popular among computer scientists and extremely problematic.
[12]For a more detailed discussion of this issue, see my "Is the Church-Turing Thesis True?" (1993, pp. 301–304).
[13]Turing introduced the concept of an "oracle" in his 1938 doctoral dissertation "Systems of logic based on Ordinals". This, work was subsequently published in 1939 in the *Proceedings of the London Mathematical Society* series (pp. 161–228).
[14]For a more detailed discussion of using an 0-machine to solve the halting problem, see Copeland (1998, pp. 129–131).
[15]I would particularly like to thank Jerry Seligman and Fred Kroon for extensive and helpful discussions of this material. I would also like to thank the Department of Philosophy in The Faculties at the Australian National University (Canberra) for providing space, equipment, and helpful discussions (particularly Peter Roeper) during the academic year 1998–99.

References

Biuso, Julia (1997), *Italian Cooking*, Newport Beach: C.J. Publishing.
Cleland, Carol E. (1993). 'Is the Church-Turing Thesis True?', *Minds and Machines* 3, pp. 283–212.
Cleland, Carol E (1995), 'Effective Procedures and Computable Functions', *Minds and Machines* 5, pp. 9–23.
Colburn, Timothy (1999), 'Software, Abstraction, and Ontology', *The Monist* 82, pp. 3–19.
Copeland, Jack (1998) 'Turing's O-machines, Searle, Penrose and the Brain,' *Analysis* 58.2, pp. 129–131.
Fetzer, James (1988), 'Program Verification: The Very Idea', *Communications of the ACM* 32, pp. 1048–1063.
Fetzer, James (1991), 'Philosophical Aspects of Program Verification? '*Minds and Machines* 1 pp. 197–216.
Goodman, Nelson (1983), *Fact, Fiction and Forecast*, Cambridge: Harvard University.
Hennie, Fred (1977), *Introduction to Computability*, Reading: Addison-Wesley.
Kripke, S. A. (1982), *Wittgenstein on Rules and Private Language*, Oxford: Blackwell.

Minsky, Marvin (1967), *Computation: Finite and Infinite Machines*, Englewood Cliffs: Prentice-Hall.
Smith, N. K., trans., *Kant's Critique of Pure Reason* (V, 1), New York: St. Martins.
Turing, Alan (1939), 'Systems of Logic based on Ordinals', *Proceedings of the London Mathematical Society* series 2, 45, pp. 161–228.