Minds & Machines (2008) 18:17–38
DOI 10.1007/s11023-007-9083-1

The Interactive Nature of Computing: Refuting the Strong Church–Turing Thesis

Dina Goldin · Peter Wegner

Received: 28 March 2006 / Accepted: 21 November 2007 / Published online: 1 January 2008

Abstract The classical view of computing positions computation as a closed-box transformation of inputs (rational numbers or finite strings) to outputs. According to the interactive view of computing, computation is an ongoing *interactive process* rather than a function-based transformation of an input to an output. Specifically, communication with the outside world happens *during* the computation, not before or after it. This approach radically changes our understanding of what is computation and how it is modeled. The acceptance of interaction as a new paradigm is hindered by the *Strong Church–Turing Thesis* (SCT), the widespread belief that Turing Machines (TMs) capture all computation, so models of computation more expressive than TMs are impossible. In this paper, we show that SCT reinterprets the original Church–Turing Thesis (CTT) in a way that Turing never intended; its commonly assumed equivalence to the original is a myth. We identify and analyze the historical reasons for the widespread belief in SCT. Only by accepting that it is false can we begin to adopt interaction as an alternative paradigm of computation. We present *Persistent Turing Machines* (PTMs), that extend TMs to capture *sequential interaction*. PTMs allow us to formulate the *Sequential Interaction Thesis*, going beyond the expressiveness of TMs and of the CTT. The paradigm shift to interaction provides an alternative understanding of the nature of computing that better reflects the services provided by today's computing technology.

Keywords Interactive computation · Church–Turing Thesis · Turing Machines · Strong Church–Turing Thesis · Persistent Turing Machines · Sequential Interaction Thesis · Computation expressiveness · Paradigm shift

D. Goldin (✉) · P. Wegner
Computer Science Department, Brown University, Box 1910, Providence, RI 02912, USA
e-mail: dqg@cs.brown.edu

Introduction

Computer science, like other disciplines, aims to describe its nature by fundamental principles that distinguish its goals and methods from those of related disciplines. Physics relied for centuries on Newton's principles of motion; likewise, computer science has been relying on Turing Machines and algorithms in the last several decades as the defining principles of our discipline.

The nature of principles evolves as the discipline evolves, and new principles replace old ones. These new principles may be seriously questioned by believers in old principles. Thus physical principles of Copernicus, Newton, and Einstein were seriously questioned when first proposed, and there is still a controversy between Einstein's relativity theory and the quantum theory of Bohr and Heisenberg. Similarly, Darwin's principles of evolution and natural selection were (and still are) questioned, in spite of their experimental validity.

Hilbert's principle that formal mathematical theorems were provable by logical inference was questioned by Godel, Turing, and Church, who argued that logic cannot completely prove all mathematical theorems. Turing's proof (Turing 1936) introduced a new model, *Turing Machines* (TMs), and showed that there are problems, such as the *Halting Problem*, that they cannot solve, despite their expressiveness. That expressiveness was captured by the *Church–Turing Thesis* (CTT), stating that TMs can compute any effective (partially recursive) function over naturals (strings). (See Cleland 2007 for a longer discussion of the history of CTT.)

While originally introduced by Turing as a tool for rejecting Hilbert's principle, TMs have since been used for establishing the principles of computer science. His thesis has been greatly misunderstood; many of the myths about it are nicely detailed in Copeland (1997). An example of an alternate thesis that is sometimes conflated with the original is the so-called *Physical Church–Turing Thesis* (PCT):

PCT: A function is effective computable by a physical system iff it is Turing machine computable.

This formulation is quoted from Fitz (1997), where PCT is discussed extensively. Copeland (1997) provides a good refutation of PCT, stressing that CTT only applies to *effective* computation (in the original sense of this term), rather than computation by arbitrary physical machines, dynamical systems, or humans (as in Cleland 2004; Sieg 2005 among others).

In the computer science community, it is generally understood that CTT applies only to effective computation, in Turing's sense of the word. However, it is not always appreciated that CTT applies only to computation of *functions*, rather than to *all* computation. Function-based computation transforms a finite input into a finite output in a finite amount of time, in a closed-box fashion. By contrast, the general notion of computation includes arbitrary procedures and processes—which may be open, non-terminating, and involving multiple inputs interleaved with outputs.

The *Strong Church–Turing Thesis* (SCT), which asserts that TMs capture *all* effective computation, is generally considered to be equivalent to the original CTT:

SCT: Every effective computation can be carried out by a TM.

This formulation can be found, among others, at the beginning of Copeland (1997), where it is claimed equivalent to the original thesis.

SCT has become a fundamental principle of computer science, defining the nature of our discipline. Stated informally, it asserts that TMs can do "everything that computers can do"; this quote is from Sipser (2005), a leading textbook for theory of computation.

All versions of the SCT historically evolve from the assumption that all computation is *function-based*, or *algorithmic*[1]; by this, we mean that its job is to transform a finite input into a finite output in a finite amount of time. We believe it is time to recognize that today's computing applications, such as web services, intelligent agents, operating systems, and graphical user interfaces, are *interactive* rather than *algorithmic*; their job is to provide ongoing services over time (Wegner 1997).

According to the interactive view of computation, interaction (communication with the outside world) happens *during* the computation, not before or after it. Hence, computation is an ongoing *process* rather than a function-based transformation of an input to an output. The interactive approach represents a paradigm shift that redefines the nature of computer science, by changing our understanding of what computation is and how it is modeled. This view of computation is not modeled by TMs, which capture only the computation of functions; alternative models are needed.

Interaction Machines are models of computation that extend TMs with interaction to capture the behavior of concurrent systems, promising to bridge the fields of computation theory and concurrency theory. We present one such model of interaction, *Persistent Turing Machines* (PTMs), originally formalized in Goldin et al. (2004). PTMs capture *sequential interactive computation* and allow us to formulate the *Sequential Interaction Thesis*, going beyond the expressiveness of TMs and of the CTT.

The idea that interaction machines are more expressive than TMs seems to question the accepted dogma, hindering its acceptance within the theoretical computer science community. However, it does not reject the original CTT, which only refers to the computation of *functions*, and thus specifically excludes interactive computation. By referring to *all* computation, SCT reinterprets the original thesis in a way that Turing never intended, and it is a fallacy to conflate the two (Goldin and Wegner 2005). When the notion of computation is extended from functions and algorithms to processes and services, SCT no longer holds.

Outline. The theoretical nature of computing is currently based on what we call *the mathematical worldview*. Section "Two Views of Computer Science" discusses this worldview, contrasting it with the *interactive worldview*. Algorithms are an old mathematical concept adopted by the early computer scientists as part of the mathematical worldview. Section "The Evolution of Algorithms" discusses how the adoption and the subsequent broadening of the notion of algorithms have led to a

[1] We use "algorithms" in their classical sense, as found in Knuth (1968).

confusion over the meaning of the Church–Turing Thesis. Section "New Models of Computation" considers what it means to model computation, and shows how to extend Turing Machines towards greater expressiveness by injecting interaction into the computation. Section "Analysis of the Strong Church–Turing Thesis" analyzes the beliefs that support the Strong Church–Turing Thesis, identifying their flaws. We conclude that the Strong Church–Turing Thesis is not equivalent to the original thesis, and therefore cannot be used to refute our claims about interactive computation. Section "The Paradigm Shift to Interaction" discusses some of the implications of the paradigm shift to interaction.

Two Views of Computer Science

The theoretical nature of computing is currently based on what we call *the mathematical worldview*. We discuss this worldview next, contrasting it with the *interactive worldview*.

The Mathematical Worldview

The theory of computation predates the establishment of computer science as a discipline, having been a part of mathematics before the 1960s. Its founders include such notable mathematicians as Godel, Kleene, Church, and Turing.[2] Mathematicians equate the notion of computability with the computation of functions. Martin Davis's 1958 textbook (Davis 1958) reflected this *mathematical worldview* that all computation is function-based, and therefore captured by Turing Machines (TMs). It begins as follows:

> "This book is an introduction to the theory of computability and non-computability, usually referred to as the theory of recursive functions... the notion of TM has been made central in the development."

In particular, this view assumes that all computation is *closed*. There is no input or output taking place during the computation; any information needed during the computation is provided at the outset as part of the input. These assumptions are embodied by the semantics of TMs.

Mathematical worldview: All computable problems are function-based.

The mathematical worldview was enthusiastically adopted by early leaders of the computer science community, including Von Neumann, Knuth, Karp, and Rabin. Mathematics has been used as a foundation of physics and other scientific disciplines, and it was believed that mathematics could be used as a basis for computer science. Davis's book proved very influential, cementing the acceptance of the mathematical worldview among computer scientists of the 1950s and 1960s.

[2] While Turing's training and original contributions were mathematical, we believe that his later work classifies him as a computer scientist rather than a mathematician—perhaps the first one.

TMs, which transform input strings to output strings, have served from the onset as a formal model for function-based computation:

A problem is *solvable* iff there exists a Turing machine for computing it.

The legitimacy of this claim is based on two premises. The first one is the *Church–Turing Thesis*, which equates function-based computation with TMs. The second one, usually left unstated, is the mathematical worldview—the assumption that all computable problems are function-based.

The batch-based modus operandi of the original computers in the 1940s and 1950s was compatible with the function-based view of computation, creating no inconsistency between the theory and the practice of the discipline at that time. The perceived validity of this claim was further strengthened by the many early attempts to find models of computation that are more expressive than TMs, for example extending the number of tapes or reading heads on the machine. All these attempts failed, because they continued to adhere to the mathematical worldview, and never considered problems that are not function-based. We return to this issue in section "Extending Turning Machines."

The Interactive Worldview

The mathematical worldview can be contrasted with the *interactive worldview*, where computation is viewed as an ongoing process that transforms inputs to outputs—e.g., control systems, or operating systems. The question "*what do operating systems compute?*" has been a conundrum for the adherents of the mathematical worldview, since these systems never terminate, and therefore never formally produce an output. Yet it is clear that they *do* compute, and that their computation is both useful and important to capture formally.

While the Church–Turing Thesis remains true, the mathematical worldview no longer reflects the nature of computational problems. An example of such a problem is *driving home from work* (Wegner 1997; Eberbach et al. 2004):

Driving home from work: create an automatic car to drive us home from work, where the locations of both work and home are provided as input parameters.

Input to this problem may include any necessary additional information (encodable as a finite string), such as a detailed map and a current weather report. The output of this problem should be two time series: one that shows the position of the wheel (e.g., in degrees from the vertical) during the drive, and another to show the pressure on the gas and break pedals during the drive.

Assuming that the driving is to take place in a real-world environment, this problem is not computable within a function-based computational paradigm. Consider the input to such a function. It must be detailed enough so the car can predict the direction and speed of the wind at every moment of the drive, so as to compensate for it. It should also enable the car to anticipate the location of all pedestrians, so as to avoid running over them. As we discuss in Eberbach et al. (2004), this is impossible—there is no computable function that will determine, given some finite amount of a priori information, all the real-world factors that are necessary to ensure the car's safe arrival at its destination. An assertion to the contrary would endow TMs with a power to predict the future that is tantamount to a resolution of the predestination debate.

However, the problem of driving home from work *is* computable—by a control mechanism, as in an robotic car, that continuously receives video input of the road and actuates the wheel and brakes accordingly. This computation, just as that of operating systems, is *interactive*, where input and output happen *during* the computation, not before or after it.

The interactive approach to conceputalizing the notion of computation and of computable problems is distinct from either the theory of computation and the concurrency theory. It represents a paradigm change to our understanding of what is computation, and how it should be modeled. This conceptualization of computation allows, for example, the *entanglement* of inputs and outputs, where later inputs to the computation depend on earlier outputs. Such entanglement is impossible in the mathematical worldview, where all inputs precede computation, and all outputs follow it.

The driving example represents an empirical proof of the claim that interactive computation is more expressive than function-based computation, i.e., it can solve a greater range of problems. However, to accept this claim, one has to broaden one's notion of a problem beyond what is prescribed by the mathematical worldview. Driving home from work, queuing jobs within an operating system, or document processing, are all legitimate computational problems on par with finding common divisors or choosing the next move on a chess board.

The Evolution of Algorithms

The notion of an *algorithm* is a mathematical concept much older than Turing Machines; perhaps the oldest example is *Euclid's algorithm* for finding common divisors. This concept is part of the mathematical worldview that was adopted by the early computer scientists. This section discusses how the adoption and the subsequent broadening of the notion of algorithms have led to a confusion over the meaning of the Church–Turing Thesis.

The Original Role of Algorithms

Algorithms originated in mathematics as "recipes" for carrying out function-based computation, that can be followed mechanically. Algorithms capture what it means for a computation to be *effective*.

Role of algorithm: Given some finite input x, an algorithm describes the steps for effectively transforming it to an output y, where y is $f(x)$ for some recursive function f.

Like mathematical formulae, algorithms tell us how to compute a value; unlike them, algorithms may involve what we now call *loops* or *branches*. Their role remained unchanged when they were adopted by the early computer scientists in the 1950s, who made the connection between algorithms and Turing Machines, equating their expressiveness.

TRUE: TMs = algorithms (in the classical, narrow sense)

Knuth's famous and influential textbook, *The Art of Computer Programming, Vol. 1* (Knuth 1968) popularized the centrality of algorithms in computer science. In his definition of algorithms, Knuth was consistent with the mathematical function-based foundations of the theory of computation. He explicitly specified that algorithms are closed; no new input is accepted once the computation has begun:

> "An algorithm has zero or more inputs, i.e., quantities which are given to it initially before the algorithm begins."

Knuth distinguished algorithms from arbitrary computation that may involve I/O. One example of a problem that is not algorithmic is the following instruction from a recipe (Knuth 1968): "toss lightly until the mixture is crumbly."[3] This problem is not algorithmic because it is impossible for a computer to know how long to mix; this may depend on external dynamically changing conditions such as humidity, that cannot be predicted with certainty ahead of time. In the function-based mathematical worldview, all inputs must be specified at the start of the computation, preventing the kind of feedback that would be necessary to determine when it's time to stop mixing. The problem of driving home from work is also among those problems that Knuth meant to exclude.

Knuth's detailed discussion of algorithmic computation ensured their function-based behavior and guaranteed their equivalence with TMs:

> "There are many other essentially equivalent ways to formulate the concept of an effective computational method (for example, using TMs)." (Knuth 1968)

While there is no agreement on the exact notion of algorithms, Knuth's discussion of algorithms remains definitive; in particular, it serves as the basis of the authors' understanding of this term.

The notion of an algorithm is inherently informal, since an algorithmic description is not restricted to any single language or formalism. The first high-level programming language developed expressly to specify algorithms was ALGOL (ALGOrithmic Language). Introduced in the late 1950s and refined through the 1960s, it was the standard for the publication of algorithms. True to the function-based conceptualization of algorithms, ALGOL provided no constructs for input and output, considering these operations outside the concern of algorithms.[4]

[3] Many similar examples can be found in Cleland's work (such as Cleland 2004).

[4] Not surprisingly, this absence hindered the adoption of ALGOL by the industry for commercial applications.

Algorithms Made Central

The 1960s saw a proliferation of undergraduate computer science (CS) programs; the number of CS programs in the USA increased from 11 in 1964 to 92 in 1968 (ACM Cirriculum 1968). This increase was accompanied by intense activity towards establishing the legitimacy of this new discipline in the eyes of the academic community. The Association for Computing Machinery (ACM) played a central role in this activity. In 1965, it enunciated the justification and description of CS as a discipline (ACM Report 1965), which served as a basis of its 1968 recommendations for undergraduate CS programs (ACM Curriculum 1968); one of the authors (Wegner) was among the primary writers of the 1968 report.

ACM's description of CS (ACM Report 1965) identified effective transformation of information as a central concern:

> "Computer science is concerned with information in much the same sense that physics is concerned with energy... The computer scientist is interested in discovering the pragmatic means by which information can be transformed."

By viewing algorithms as transformations of input to output, ACM adapted an algorithmic approach to computation; this is made explicit in the next sentence of the report:

> "This interest leads to inquiry into effective ways to represent information, effective algorithms to transform information, effective languages with which to express algorithms...and effective ways to accomplish these at reasonable cost."

Having a central algorithmic concern, analogous to the concern with energy in physics, helped to establish CS as a legitimate academic discipline on a par with physics.

Algorithms have remained central to computer science to this day. The coexistence of the informal (algorithm-based) and the formal (TM-based) approaches to defining solvable problems persists to this day and can be found in all modern textbooks on algorithms or computability:

- A problem is *solvable* if it can be specified by an algorithm.
- A problem is *solvable* if there exists a Turing Machine for it.

This has proved very convenient for computer scientists, allowing us to describe algorithmic computation informally using "pseudocode", with the knowledge that an equivalent Turing Machine can be constructed.

Algorithms Redefined

The 1960s decision by theorists and educators to place algorithms at the center of CS was clearly reflected in early undergraduate textbooks. However, there was no explicit standard definition of an algorithm and various textbooks chose to define this term differently. While some textbooks (such as Knuth 1968) were careful to explicitly restrict algorithms to those that compute functions and are therefore TM-equivalent, most theory books left this restriction implied but unstated.

An early example is Hopcroft and Ullman (1969), one of the first textbooks on the theory of computation (whose later editions are being used to this day):

> "A procedure is a finite sequence of instructions that can be mechanically carried out, such as a computer program... A procedure which always terminates is called an algorithm."

Their discussion of algorithms does not *explicitly* preclude non-functional computation such as driving home from work. However, the prohibition against obtaining inputs dynamically during the computation is *implicitly* present. Their examples of various problems make it clear that only function-based computation was considered. In fact, ALGOL, the language then used for writing algorithmic programs, did not even offer any constructs for input and output.

By the late 1960s, the high-level programming languages used in practice and taught in CS programs were no longer bound by the algorithmic restraints of early ALGOL. In response, contemporary programming textbooks (such as Rice and Rice 1969) explicitly broadened the notion of algorithms to include non-functional problems. This approach, reflecting the centrality of algorithms without restricting them to the computation of functions, is typical of non-theory textbooks.

On the surface, the definition of an algorithm in Rice and Rice is no different from Hopcroft and Ullman:

> "An algorithm is a recipe, a set of instructions or the specifications of a process for doing something. That something is usually solving a problem of some sort."

However, their examples of computable problems are no longer function based, admitting just the kind of computation that Knuth had rejected. Two such examples, that can supposedly be solved by an algorithm, are making potato vodka and filling a ditch with sand; driving home from work would fit right in, too.

TRUE: computable problems = algorithms (in the common, broader sense)

The subject of Rice and Rice was programming methodology rather than the theory of computation, and the mathematical principles that underpin our models of computation were cast aside for the sake of practicality. Rice and Rice made no claims of TM-equivalence for their "algorithms". However, the students were not made aware that this notion of algorithms is different from Knuth's, and that the set of problems considered computable had thereby been enlarged.

By pairing a broader, more practical, conceptualization of algorithms (and hence of computable problems) with theories claiming that every computable problem can be computed by TMs, the algorithm-focused CS curriculum left students with the erroneous impression that this broader set of problems could also be solved by TMs.

FALSE: TMs = computable problems

Algorithms Today

A recent ACM SIGACT newsletter acknowledges that of all undergraduate CS subjects, theoretical computer science has changed the least over the decades (SIGACT News 2004). While the practical computer scientists have long since followed the lead of Rice and Rice (1969) and broadened the concept of algorithms beyond the computation of functions, theoretical computer science has retained the mathematical worldview that frames computation as function-based, and delimits our notion of a computational problem accordingly. This is true at least at the undergraduate level, despite advanced complexity theoretic work that ventures outside this worldview, such as on-line and distributed algorithms, Arthur–Merlin games, and interactive proofs.

The result is a dichotomy, where the computer science community thinks of algorithms as synonymous with the general notion of computation ("what computers do") yet at the same time as being equivalent to Turing Machines. This dichotomy can be found in today's popular textbooks (such as Sipser 2005). Their discussion of algorithms is very broad, but the equivalence with TMs is taken for granted:

> "an algorithm is a collection of simple instructions for carrying out some task. Commonplace in everyday life, algorithms sometimes are called procedures or recipes... The TM merely serves as a precise model for the definition of algorithm."

While the selection of computational problems in Sipser (2005) is all function-based, this discussion of algorithms certainly leaves an impression that tasks such as operating system processes are considered algorithmic. After all, these are tasks that computers carry out all the time.

This dichotomy is canonized in the Strong Church–Turing Thesis, which can be found throughout the computing literature, including Sipser (2005):

> "A TM can do anything that a computer can do."

New Models of Computation

We now consider what it means to model computation, and how to extend Turing Machines towards greater expressiveness by injecting interaction into the computation.

Extending Turing Machines

The history of modifying or extending TMs is at least as old as the theory of computation. By TMs, we mean Turing's *automatic machines* as defined in his original paper (Turing 1936), and all the versions of these machines that are equivalent to the original. In general, the equivalence of TM versions cannot be taken for granted. As discussed in section "Syntax versus Semantics", the versions one obtains by modifying or extending TMs are *not* TMs, unless and until equivalence with the original has been proven. For example, Turing's automatic machines had a binary alphabet and an infinite tape; the Turing Machine we use now typically has an arbitrary alphabet and a semi-infinite tape. Before this version could be called a Turing Machine, a proof was needed of the equivalence of the two models. Indeed, if only right transitions are allowed in a TM, the resulting model is not equivalent, having the expressiveness of a finite state automaton (FSA) rather than a TM.

The FSA example represents a *restriction* on TM computations. More common are *extensions*. All TM extensions that can be found in theory textbooks, such as increasing the number of tapes or changing the alphabet, are algorithmic. In the case of algorithmic extensions, the Church–Turing Thesis applies, and it can be taken for granted that the new model is equivalent to the original. However, as a result of the Strong Church–Turing Thesis, we have come to assume the equivalence of *any* TM extension (algorithmic or not) to the original, and we no longer require (or expect) formal proofs of this.

To capture the contemporary interactive use of computers, the more recent TM extensions have tended to be non-algorithmic, with computation that spans multiple inputs and outputs to the underlying TM. Nowadays we consider non-terminating interactive computations of Turing Machines, persistent Turing Machines, nets of Turing Machines, Turing Machines with evolvable architecture, etc., exactly for reasons of capturing "all computers", or "all computations".

While these extensions usually share TM syntax (machinery), their semantics are different; for example, in the case of Persistent Turing Machines, it is based on dynamic input streams and persistence (section "PTMs: Modeling Sequential Interaction"). Out of habit, researchers have continued to assume that these new machines must be equivalent to the original TM. But in the case of such non-algorithmic extensions, Turing's thesis does not automatically apply, and equivalence can no longer be taken for granted; indeed, it no longer holds.

PTMs: Modeling Sequential Interaction

Wegner has conjectured (Wegner 1997, 1998) that interactive models of computation are more expressive than "algorithmic" ones such as Turing Machines. In other words, they are able to solve a larger set of computable problems, or to exhibit a greater variety of behaviors. It would therefore be interesting to see what minimal extensions are necessary to Turing Machines to capture the salient aspects of interactive computing towards proving Wegner's conjecture. This question served as a motivation for a new model of computation called *Persistent Turing Machines* (PTMs), introduced by one of the authors (Goldin et al. 2004).

PTMs are interaction machines that extend Turing Machine semantics in two different ways, with *dynamic streams* and *persistence*, capturing sequential interactive computations. A PTM is a nondeterministic 3-tape Turing Machine (N3TM) with a read-only input tape, a read/write work tape, and a write-only output tape. Its input is a stream of *tokens* (strings) that are generated dynamically by the PTM's *environment* during the computation.

A PTM computation is an infinite sequence of *macrosteps*; the *i*th macrostep consumes the *i*th input token a_i from the input stream, and produces the *i*th output token o_i for the output stream. Each macrostep is an N3TM computation consisting of multiple N3TM transitions (microsteps), just as each input and output token is a string consisting of multiple characters. The input and output tokens are temporally interleaved, resulting in the *interaction stream* $\{(a_1, o_1), (a_2, o_2),\}$. This stream represents the observed behavior of the PTM during the computation.

A simple example of a PTM is an answering machine:

Answering Machine: An *answering machine AM* is a PTM whose work tape contains a sequence of recorded messages and whose operations are `record`, `play` and `erase`. The behavior of *AM* is defined as follows:

- when the input is `record` *Y*, and the tape contains *X*, the output is `ok`, and the tape now contains *XY*;
- when the input is `play` and the tape contains *X*, the output is *X*, and the tape now contains *X*;
- when the input is `erase` and the tape contains *X*, the output is `done` and the tape is now blank (contains an empty string).

For example, given the input stream [`record A`, `erase`, `record BC`, `record D`, `play`, ...], *AM* generates the output stream [`ok`, `done`, `ok`, `ok`, `BCD`, ...], resulting in the following interaction stream:

[(`record A`, `ok`), (`erase`, `done`), (`record BC`, `ok`), (`record D`, `ok`), (`play`, `BCD`), ...]

Note that both the content of the work tape and the length of input for recorded messages are unbounded.

The automatic car of section "The Interactive Worldview" serves as another example of a PTM. Its inputs consist of images of the road from a built-in camera, coupled with information streaming in from other parts of the car, such as the wheels. The outputs consist of signals to the steering wheel (how much to turn right or left), and to the breaks and the gas (how much to speed up or slow down). The car exemplifies I/O *entanglement*: what the car "sees" next depends on where it turns now.

PTM computations are *persistent* in the sense that a notion of "memory" (work-tape contents) is maintained from one macrostep to the next. Thus the output of each macrostep o_i depends both on the input a_i and on the work tape contents at the beginning of the macrostep. However, this contents is hidden internally, and is not considered *observable*. Thus it is not part of interaction streams, which only reflect input and output (observable) values.[5] An analogy is a human whose behavior we can observe, but whose thoughts and memories are hidden from us. While humans are not machines, humanoid robots certainly are, and PTMs can model them analogously to the way they model robotic cars.

[5] While this seems like a minor detail, it plays a key role in PTM theory, such as when defining PTM equivalence or constructing PTM simulations, as in a universal PTM (Goldin et al. 2004).

Expressiveness of Sequential Interaction

Three results concerning the expressiveness of PTMs are discussed next. We leave out the proofs, which are very technical and are outside the scope of this paper; they can be found in Goldin et al. (2004).

The first result is that the class of PTMs is isomorphic to *interactive transition systems* (ITSs), which are effective transition systems whose actions consist of input/output pairs—thereby allowing one to view PTMs as ITSs “in disguise”. This result addresses an open question concerning the relative expressive power of Turing Machines and transition systems. It has been known that transition systems are capable of simulating Turing Machines. The other direction, namely “What extensions are required of Turing Machines so they can simulate transitions systems?”, is solved by PTMs.

The second result is the greater expressiveness of PTMs over *Amnesic Turing Machines* (ATMs), which are a subclass of PTMs that do not have persistence—in effect by erasing their work tape. ATMs extend Turing Machines with dynamic streams but without memory. An example of an ATM is a *squaring machine*, whose input and output are streams of numbers; at ith macrostep, if the input number is k_i, the output is its square k_i^2.

Persistence extends the effect of inputs. An input token affects the computation of its corresponding macrostep, including the work tape. The work tape in turn affects subsequent computation steps. If the work tape were erased, then the input token could not affect subsequent macrosteps, but only “its own” macrostep. With persistence, an input token can have impact on all subsequent macrosteps; this property of PTM computations is known as *history dependence*. By contrast, ATM computations exhibit no history dependence; intuitively, an ATM computation is a series of separate computations of the same TM, reinitializing it before each new input. The strictly greater expressiveness of PTMs over ATMs also implies that PTMs are more expressive than Turing Machines, allowing us to formally prove Wegner’s conjecture.

The third result proves the existence of *universal PTMs*; analogously to a universal Turing Machine, a universal PTM can simulate the behavior of any arbitrary PTM.

Sequential Interaction Thesis

PTMs perform *sequential interactive* computation, defined as follows:

Sequential Interactive Computation: A sequential interactive computation continuously interacts with its environment by alternately accepting an input string and computing the corresponding output string. Each output-string computation may be both nondeterministic and history-dependent, with the resultant output string depending not only on the current input string, but also on all previous input strings.

Examples of sequential interactive computation abound, including Java objects, static C routines, single-user databases, and network protocols. A *simulator PTM* can be constructed for each of these examples, analogously to the construction of the universal PTM. The result is a sequential interactive analogue to the Church–Turing Thesis, stating that PTMs capture all sequential interaction:

Sequential Interaction Thesis: Any sequential interactive computation can be performed by a Persistent Turing Machine.

This thesis establishes the foundation of the theory of sequential interaction, with PTMs and ITSs as its alternative canonical models of computation. Since PTMs are more expressive than amnesic TMs and Turing Machines, the theory of sequential interaction represents a more powerful problem-solving paradigm than the traditional theory of computation (TOC), confirming the conjecture that "interaction is more powerful than algorithms". We also expect that this theory will prove as robust as TOC, with appropriate analogues to fundamental TOC concepts such as logic, complexity, and recursive functions.

Analysis of the Strong Church–Turing Thesis

In this section, we analyze the beliefs that support the Strong Church–Turing Thesis, identifying their flaws. Our goal is to show that the Strong Church–Turing Thesis is not equivalent to the original thesis, and therefore cannot be used to refute our claims about interactive computation.

Confusion over Algorithms

The Church–Turing Thesis, developed when Turing visited Church in Princeton in 1937–1938, asserted that Turing Machines and the lambda calculus could compute all effective (recursive) mathematical functions.

Church–Turing Thesis: Whenever there is an effective method for computing a mathematical function it can be computed by a Turing machine or by the lambda calculus.

This thesis identifies the notion of *effective computability of functions* over natural numbers (strings) with the computation of Turing Machines. Church and Turing showed that this notion of effective computability, based on transformations of inputs to outputs, could also be realized by the Lambda Calculus and by recursive functions.

The literature is replete with various reinterpretations of CTT, such as the *Physical Church-Turing Thesis* (PCT), which extends: the effectiveness of function computation to the physical realm:

PCT: A function is effective computable by a physical system iff it is Turing machine computable.

Computer science literature has embraced a different reinterpretation. The Church–Turing Thesis is viewed by the computer science community as implying that Turing Machines model *all* effective computation, rather than just functions. We call this claim the *Strong Church-Turing Thesis*:

Strong Church–Turing Thesis: A Turing machine can compute anything that any computer can compute. It can solve all problems that are expressible as computations (well beyond computable functions).

While the Church–Turing Thesis is correct, this later version is not equivalent to it; Turing himself would have denied it. It is not well known that in the same famous paper where he introduced TMs (calling them *automatic machines*, or *a-machines*) he also introduced *choice machines* (*c-machines*), which extend TMs to interaction by allowing a human operator to make choices during the computation (Turing 1936). It is clear that Turing viewed c-machines as another model of computation distinct from TMs and not reducible to it.

In fact, the Strong Church–Turing Thesis is incorrect—the function-based behavior of algorithms does not capture all forms of computation, and Persistent Turing Machines are provably more expressive than Turing Machines (section "PTMs: Modeling Sequential Interaction"). Since this thesis is not equivalent to the original, a proof of its incorrectness does not challenge the original thesis. However, the Strong Church–Turing Thesis is still dogmatically accepted by most theoretical computer scientists as a mathematical principle for computing that underlies computer science. As Denning recently wrote (Denning 2004), "we are captured by a historic tradition that sees programs as mathematical functions".

The equivalence of the Strong Church–Turing Thesis to the original is a myth, whose widespread acceptance rests on the following claims:

Claim 1. All computable problems are expressible by functions from integers to integers, and therefore captured by Turing Machines.
Claim 2. All computable problems can be described by algorithms.
Claim 3. Algorithms are what computers do.

The first of these claims reflects *the mathematical worldview*, discussed in section "The Mathematical Worldview." It is based on the Church–Turing Thesis, which equates functions and Turing Machines. The third claim reflects the central position of algorithms in defining practical computation, discussed in section "Algorithms Redefined." The second claim ties the first and the third claims together, creating an appearance of equivalence between Turing Machines and computers.

As discussed in section "The Evolution of Algorithms", there exist two distinct interpretations of the notion of algorithms (and correspondingly, of computing problems). The first is the *classical* interpretation, which defines algorithms as function-based transformations of inputs to outputs. The second interpretation is *pragmatic*, defining algorithms as abstract descriptions of the behaviors of programs. The first interpretation of claim 2 is compatible with claim 1, while the second interpretation is compatible with claim 3; however, these interpretations of claim 2 are mutually incompatible and cannot coexist. This incompatibility pulls apart the three claims, bringing down the Strong Church–Turing Thesis.

Syntax versus Semantics

As discussed above, the confusion over the notion of algorithms and the coexistence of two mutually incompatible interpretations played a key role in engendering the myth of the Strong Church–Turing Thesis. However, other factors also played a role, by "corroborating" the correctness of this thesis. One of them is the failure to distinguish between the *syntax* and the *semantics* of automata, such as the Turing Machine (TM).

TM syntax: what does it consist of?

TM semantics: how does it compute?

A TM *consists of* a finite set of states, a read/write head, a tape, and a control mechanism for transitioning between states and performing read/write actions on the tape. At this level, the description of a TM is similar to that of a computer. The differences, as pointed out in Wegner (1968), are that the computer's memory is not infinite, and it is accessed randomly rather than sequentially. But these differences are relatively minor, and the following claim has been made:

TMs serve as a general model for computers.

This claim, which can be viewed as corroborating the Strong Church–Turing Thesis, focuses on TM syntax. However, just as for any other class of automata, TMs are not fully specified until we define what is meant by their computation, i.e., their semantics. As defined by Turing in 1936, TM semantics prescribe that every computation starts in an identical configuration (except for the contents of the read/write tape), and the contents of the tape cannot be modified from the "outside" during the computation. This can be contrasted with Turing's alternate model of *choice machines* (Turing 1936), which has the same syntax as TMs but different semantics.

Statements about TM expressiveness, such as the Church–Turing Thesis, fundamentally depend on their semantics. If these semantics were defined differently, it may (or may not) produce an equivalent machine. Kugel uses the terms *machinery* and *machines* to describe the same distinction between "what a machine contains" and "how it is used" (Kugel 2002). He points out that if used differently, the same machinery results in a different machine. An example of a model that shares TM's syntax (machinery) but has different semantics are Persistent Turing Machines; its semantics are based on *dynamic streams* and *persistence* (section "PTMs: Modeling Sequential Interaction").

The computation of early computers could be said to reflect TM semantics. However, the computation of modern computers is no longer based on the same semantics. Unlike TMs, new inputs arrive continuously (think of an operating system, or a document processor such as Word or PowerPoint); the output is also produced continuously (in case of the document processor, it is the screen display of the document). There is *I/O entanglement*, whereby later inputs are affected by earlier outputs and vise-versa. All this renders a computer's behavior *non-functional*; it no longer computes a function from the input to the output, and TM no longer serves as an appropriate model for this *interactive* computational behavior.

The Universal Turing Machine

The *Universal Turing Machine* is a special TM introduced by Turing in 1936, that can simulate any other TM. Given the complete description of any TM M and some input string w, the universal TM U will return the output of M on W:

$$U(M, w) = M(w)$$

This machine served as the inspiration for the notion of *general-purpose computing*. Turing himself saw a direct parallel between the capability of a computer to accept and execute programs, and his notion of a universal machine.

The principle of universality can easily be extended to any other class of algorithmic machines:

Universality Thesis: Any class of effective devices for computing functions can be simulated by a TM.

As discussed in section "Syntax versus Semantics", the semantics of a TM only allow it to simulate devices that compute functions, such as M above. M cannot be a computer, because a computer's complete description (including its memory) changes with every input. After we simulate a computer C on input w_1, we obtain in essence a new computer C'. To continue the simulation on input w_2, we need to simulate C' rather than C. By contrast, U must start its simulation from the same configuration for each input.

This inconvenient restriction is absent from real-world simulations, such as when running an interpreter. The failure to appreciate the distinction between the notion of simulation as experienced in practice and as prescribed for Turing Machines has led to the strengthening of the Universality Thesis:

Strong Universality Thesis: Any computer can be simulated by a TM.

The Strong University Thesis has also been used to "corroborate" the Strong Church–Turing Thesis, but its foundations are equally questionable.

The Strong Church–Turing Thesis Refuted

Earlier, we have discussed the origins for the widespread belief in the Strong Church–Turing Thesis, having identified three distinct claims that lead to it:

Claim 1. All computable problems are function-based.
Claim 2. All computable problems can be described by an algorithm.
Claim 3. Algorithms are what computers do.

We have also looked at two more claims that have been used to corroborate the myth of the Strong Church–Turing Thesis

Claim 4. TMs serve as a general model for computers.
Claim 5. TMs can simulate any computer.

Each of these claims contains a grain of truth. By reformulating them to bring out the hidden assumptions, misunderstandings are removed. The following versions of the above statements are correct:

New claim 1. All algorithmic problems are function-based.
New claim 2. All function-based problems can be described by an algorithm.
New claim 3. Algorithms are what early computers used to do.
New claim 4. TMs serve as a general model for early computers.
New claim 5. TMs can simulate any algorithmic computing device.

Furthermore, the following claim is also correct:

Claim 6. TMs cannot compute all problems, nor can they do everything that real computers can do.

This last claim, while incompatible with original five claims, is perfectly consistent with their corrected versions. By contradicting the Strong Church–Turing Thesis, it legitimizes the search for models of computation that are more expressive than TMs, without challenging the original Church–Turing Thesis.

The Paradigm Shift to Interaction

This section discusses some of the implications of the paradigm shift to interaction.

Expressiveness of Interaction

Sequential interaction is only one form of interactive computation. The paper "Interactive Foundations of Computing" by one of the authors (Wegner 1998) explores more general notions of interaction, including analog, real-time, and multi-agent interaction. It conjectures that these forms of interaction are more expressive than sequential interaction.

It shows that the semantics of streams cannot be expressed by that of strings, that interaction includes nonfunctional non-algorithmic behavior, that persistent agents are not algorithmically describable, that extending algorithms to interaction transforms "dumb" to "smart" problem-solving behaviors. It furthermore shows that increased expressive power may in many cases decrease or eliminate *formalizability*—the ability to state a problem (or describe a computing system) formally and completely.

Dijkstra's "GOTO considered harmful" article (Dijkstra 1968) suggested eliminating the GOTO statement because its expressiveness decreased formalizability. Interaction likewise increases expressiveness at the expense of formalizability, and could therefore be considered harmful in the Dijkstra sense. We believe that expressiveness should be encouraged and that neither interaction nor GOTO statements should be considered harmful because they decrease formalizability.

Interactive systems are incomplete in the sense that they cannot be modeled by sound and complete first-order logics. Incompleteness contributes to expressiveness at the expense of formalizability, supporting non-formalizable error checking, emergent behavior, open systems, object-oriented programming, and robustness. Programming in the large produces non-formal interactive behaviors that are more expressive than algorithmic programming in the small.

The paradigm shift from algorithms to interaction suggests that mathematical formalizability must necessarily be restricted in realizing important goals of expressiveness. Interactive models extend formal *rationalist* systems that limit expressiveness to non-formal *empiricist* systems that are more expressive (Wegner 1999). Thus rationalist mathematical arguments that formalizability is an essential tool of problem solving must be eliminated in achieving the broader goal of empiricist interactive problem solving.

Interaction and Concurrency

Interactive computation is inherently *concurrent*, where the computation of interacting agents or processes proceeds in parallel. Hoare, Milner and other founders of *concurrency theory* have long realized that TMs do not model all of computation (Wegner and Goldin 2003). However, when their theory of concurrent systems was first developed in the late 1970s, it was premature to openly challenge TMs as a complete model of computation. Their theory positions *interaction* as orthogonal to *computation*, rather than a part of it. By separating interaction from computation, the question whether the models for CCS and the π-calculus went beyond Turing Machines and algorithms was avoided.

The resulting divide between the theory of computation and concurrency theory runs very deep. The theory of computation views computation as a *closed-box* transformation of inputs to outputs, completely captured by Turing Machines. By contrast, concurrency theory focuses on the *communication* aspect of computing systems, which is not captured by Turing Machines—referring both to the communication between computing components in a system, and the communication between the computing system and its environment. As a result of this division of labor, there has been little in common between these fields and their communities of researchers. According to Papadimitriou (1995), such a disconnect within the theory community is a sign of a crisis and a need for a Kuhnian paradigm shift in our discipline.

In the last three decades, computing technology has shifted from mainframes and microstations to networked embedded and mobile devices, with the corresponding shift in applications from number crunching and data processing to the Internet and ubiquitous computing. We believe it is no longer premature to encompass interaction *as part of* computation. Our approach is distinct from both concurrency theory and the theory of computation.

Persistent Turing Machines (PTMs, section "PTM's Modeling Sequential Interaction") are interactive machines that begin to bridge the gap between the two fields. PTMs offer a model based on the methods of the theory of computation, that captures sequential interaction, which is a limited form of concurrency. They can also be viewed as *interactive transition systems*, with corresponding notions of observational equivalence. Furthermore, they have been shown to be more expressive than Turing Machines. It is hoped that PTMs will contribute to a new theory of interactive computation which will bridge the current theories of computation and concurrency.

Conclusion

Throughout the years, researchers in theoretical computer science have found need for interactive models of computation. In addition to the work in concurrency (section "Interaction and Concurrency"), models from the 1970s and 1980s include on-line TMs for on-line algorithms (Fischer and Stockmeyer 1974), Input/Output automata for distributed algorithms (Lynch and Tuttle 1989), and Interactive TMs for interactive proofs (Goldwasser et al. 1989). However, the issue of the greater expressiveness of interactive models over TMs was not raised until the mid-1990s, when the model of interaction machines was first proposed by one of the authors (Wegner 1997).

The theoretical framework for PTMs, sequential interaction machines that are a persistent stream-based extension to TMs, was completed by one of the authors in Goldin et al. (2004); it is discussed in section "PTM's Modeling Sequential Interaction." Van Leeuwen, a Dutch expert on the theory of computation, proposed an alternate extension in van Leeuwen and Wiedermann (2000). In addition to interaction, other ways to extend computation beyond Turing Machines have been considered. *Hypercomputation* is the computation of functions that are not Turing-computable; a survey of hypercomputation can be found in Copeland (2002).

The field of artificial intelligence (AI) has perhaps gone the furthest in explicitly recognizing the expressiveness gains of moving beyond algorithms. In the early 1990s, Rodney Brooks convincingly argued against the algorithmic approach of "good old-fashioned AI", positioning interaction as a prerequisite for intelligent system behavior (Brooks 1991). This argument has been adopted by the mainstream AI community, whose leading textbooks recognize that *interactive agents* are a stronger model of intelligent behaviors than simple input/output functions (Russell and Norveig 1994).

Interaction represents a paradigm shift that changes our understanding of what is computation and how it is modeled. We recognize that our proposed paradigm shift may be questioned by proponents of the old paradigm, on the basis that Turing Machines in fact express interactive as well as other computation and that the well known computer theorists of the 1960s and 1970s have confirmed the correctness of the old paradigm. These claims constitute what we call the *Strong Church–Turing Thesis*.

We have shown that the Strong Church-Turing Thesis departs from the original Church–Turing Thesis and is incorrect in its claims. Turing's model, which only captured the closed-box computation of functions and algorithms, never claimed to model all computation, which includes non-algorithmic processes that interact during computation. The Strong Church–Turing Thesis has been a myth that negated Turing's fundamental beliefs when it was formulated 10 years after his death.

It is natural that any paradigm shift will be questioned and that arguments for a new paradigm must be verified and widely tested before it can be accepted by a broader community. We hope that this paper contributes to the understanding of the origins of the pervasive myth of the Strong Church–Turing Thesis, and to the acceptance of the interaction paradigm as an extension and replacement of the Strong Church–Turing Thesis by a broader unifying principle for computation and problem solving. Interaction provides a new conceptualization of the nature of computing that is more appropriate for modeling the services provided by the computing technology of the new millennium.

Acknowledgements We sincerely thank the anonymous reviewers for their comments, which were very helpful in revising this paper.

References

ACM Curriculum Committee on Computer Science. (1965). An undergraduate program in computer science—preliminary recommendations. *Communications of the ACM, 8*(9), 543–552.

ACM Curriculum Committee on Computer Science. (1969). Curriculum 68: Recommendations for academic programs in computer science. *Communications of the ACM, 11*(3), 151–197.

Brooks, R. (1991). *Intelligence without reason*. Technical Report 1293. MIT Artificial Intelligence Lab. Cambridge, MA.

Cleland, E. C. (2004). The concept of computability. *Theoretical Computer Science, 317*(1–3), 209–225.

Cleland, E. C. (2007). In A. Olszewski (Eds.), *The Church–Turing Thesis: A last vestige of a failed mathematical program* (pp. 119–149).

Copeland, B. J. (1997). The Church–Turing Thesis. *Stanford Encyclopedia of Philosophy* (substantially revised in 2005).

Copeland, B. J. (2002). Hypercomputation. *Minds and Machines, 12*(4), 461–502.

Davis, M. (1958). *Computability & unsolvability*. McGraw-Hill.

Denning, P. (2004). The field of programmers myth. *Communications of the ACM*, *47*(7).

Dijkstra, E. W. (1968). Go to statement considered harmful. *Communications of the ACM, 11*(3), 147–148.

Eberbach, E., Goldin, D., & Wegner, P. (2004). Turing's ideas and models of computation. In C. Teuscher (Ed.), *Alan turing: Life and legacy of a great thinker*. Springer.

Fischer, M. J., & Stockmeyer, L. J. (1974). Fast on-line integer multiplication. *Journal of Computer and System Sciences, 9*(3), 317–331.

Fitz, H. (2007). In A. Olszewski, et al. (Eds.), *Church's thesis and physical computation* (pp. 175–219).

Goldin, D., Smolka, S., Attie, P., & Sonderegger, E. (2004). Turing Machines, transition systems, and interaction. *Information & Computation Journal*, *194*(2), 101–128.

Goldin, D., & Wegner, P. (2005). *The Church–Turing Thesis: Breaking the myth*. LNCS 3526 (pp. 152–168). Springer.

Goldwasser, S., Micali, S., & Rackoff, C. (1989). The knowledge complexity of interactive proof systems. *SIAM Journal of Computing, 18*(1), 186–208.

Hopcroft, J. E., & Ullman, J. D. (1969). *Formal languages and their relation to automata*. Addison-Wesley.

Knuth, D. (1968). *The art of computer programming, Vol. 1: Fundamental algorithms*. Addison-Wesley.

Kugel, P. (2002). Computing machines can't be intelligent (...and Turing said so). *Minds and Machines*, *12*(4), 563–579.

Lynch, N. A., & Tuttle M. R. (1989). An introduction to input/output automata. *CWI Quarterly, 2*(3), 219–246. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.
Olszewski, A., & Wolenski, J., et. al. (Eds.) (2006). *Church's thesis after 70 years*. Ontos-Verlag.
Papadimitriou, C. H. (1995). Database metatheory: Asking the big queries. In *Proceedings of the 14th ACM Symposium on Principles of Database Systems*, San Jose, CA.
Rice, J. K., & Rice J. R. (1969). *Introduction to computer science: Problems, algorithms, languages, information and computers*. USA: Holt, Rinehart and Winston.
Russell, S., & Norveig, P. (1994). *Artificial intelligence: A modern approach*. Addison-Wesley.
SIGACT News. (2004). ACM Press, p. 49.
Sieg, W. (2005). *Computability and discrete dynamical systems*. LNCS 3526 (pp. 440–440). Springer.
Sipser, M. (2005). *Introduction to the theory of computation* (2nd ed.). PWS Publishing Company.
Turing, A. (1936). On computable numbers, with an application to the Entscheidungs problem. *Proceedings of the London Mathematical Society, 42*(2), 230–265; A correction, ibid, *43*, 544–546.
van Leeuwen, J., & Wiedermann, J. (2000). The turing machine paradigm in contemporary computing. In B. Enquist & W. Schmidt (Eds.), *Mathematics unlimited—2001 and beyond*. LNCS. Springer-Verlag.
Weger, P. (1968). *Programming languages, information structures and machine organization*. McGraw-Hill.
Wegner, P. (1997). Why interaction is more powerful than algorithms. *Communications of the ACM*, *40*, 80–91.
Wegner, P. (1998). Interactive foundations of computing. *Theoretical Computer Science*, *19*(2), 315–351.
Wegner, P. (1999). Towards empirical computer science. *The Monist*, *82*(1), 58–108.
Wegner, P., & Goldin, D. (2003). Computation beyond Turing Machines. *Communications of the ACM*, *46*, 100–102.