The Church-Turing Thesis: Breaking the Myth

Dina Goldin[1] and Peter Wegner[2]

[1] University of Connecticut, Storrs, CT, USA
dqg@cse.uconn.edu
[2] Brown University, Providence, RI, USA
pw@cs.brown.edu

Abstract. According to the interactive view of computation, communication happens *during* the computation, not before or after it. This approach, distinct from concurrency theory and the theory of computation, represents a paradigm shift that changes our understanding of what is computation and how it is modeled. Interaction machines extend Turing machines with interaction to capture the behavior of concurrent systems, promising to bridge these two fields. This promise is hindered by the widespread belief, incorrectly known as the Church-Turing thesis, that no model of computation more expressive than Turing machines can exist. Yet Turing's original thesis only refers to the *computation of functions* and explicitly excludes other computational paradigms such as interaction. In this paper, we identify and analyze the historical reasons for this widespread belief. Only by accepting that it is false can we begin to properly investigate formal models of interaction machines. We conclude the paper by presenting one such model, *Persistent Turing Machines* (PTMs). PTMs capture *sequential interaction*, which is a limited form of concurrency; they allow us to formulate the *Sequential Interaction Thesis*, going beyond the expressiveness of Turing machines and of the Church-Turing thesis.

1 Introduction

The fields of the theory of computation and concurrency theory have historically had different concerns. The theory of computation views computation as a *closed-box* transformation of inputs to outputs, completely captured by Turing machines (TMs). 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.

According to the interactive view of computation, communication (input/output) happens *during* the computation, not before or after it. This approach, distinct from either concurrency theory or the theory of computation, represents a paradigm shift that changes our understanding of what computation is and how it is modeled [Weg97]. Interaction machines extend Turing machines with interaction to capture the behavior of concurrent systems, promising to bridge these two fields.

The idea that Turing machines do not capture all computation, and that interaction machines are more expressive, seems to fly in the face of accepted dogma, hindering its acceptance within the theory community. In particular, the Church-Turing thesis is commonly interpreted to imply that Turing machines model all computation. It is a myth that the original thesis is equivalent to this interpretation of it. In fact, the Church-Turing thesis only refers to the computation of *functions*, and specifically excludes interactive computation. The original contribution of this paper is to identify and analyze the historical reasons for this myth.

It is time to recognize that today's computing applications, such as web services, intelligent agents, operating systems, and graphical user interfaces, cannot be modeled by Turing machines; alternative models are needed. Only by facing the fact that this reinterpretation is a myth can we begin to properly investigate these alternative models. We present one such model, *Persistent Turing Machines* (PTMs), originally formalized in [GSAS04]. PTMs capture *sequential interaction*, which is a limited form of concurrency; they allow us to formulate the *Sequential Interaction Thesis*, going beyond the expressiveness of Turing machines and of the Church-Turing thesis.

2 The Turing Thesis Myth

Turing's famous 1936 paper [Tur36] developed the *Turing Machine* (TM) model and showed that TMs have the expressiveness of algorithms (now known as the Church-Turing Thesis).

Church-Turing Thesis: Whenever there is an effective method (algorithm) for obtaining the values of a mathematical function, the function can be computed by a TM.

TMs are identified with the notion of *effectiveness*; *computability* is the current term for the same notion. Specificially, they capture computable functions as effective transformations from finite strings (natural numbers) to finite strings.

The Church-Turing thesis has since been reinterpreted to imply that Turing Machines model *all* computation, rather than just functions. This claim, which we call the *Strong Church-Turing Thesis*, is part of the mainstream theory of computation. In particular, it can be found in today's popular undergraduate theory textbooks:

Strong Church-Turing Thesis: A TM can do (compute) anything that a computer can do.

It is a myth that the original Church-Turing thesis is equivalent to this interpreation of it; Turing himself would have denied it. In the same famous paper, he also introduced interactive *choice machines* as another model of computation distinct from TMs and not reducible to it. Choice machines extend TMs to interaction by allowing a human operator to make choices during the computation.

In fact, the Strong Church-Turing Thesis is incorrect – the function-based behavior of algorithms does not capture all forms of computation. For example, as explained in [Weg97], interactive protocols are not algorithmic. Yet the myth of the correctness of the Strong Turing Thesis is dogmatically accepted by most computer scientists. As Denning recently wrote [Den04], "we are captured by a historic tradition that sees programs as mathematical functions".

The reasons for the widespread acceptance of what we call the "Turing Thesis myth" can be traced to the establishment of computer science as a separate discipline in the 1960's. To understand these reasons better, we can identify three distinct claims that make up the myth, and examine them individually:

Claim 1. All computable problems are function-based.
Reason: Adoption of mathematical principles for the fundamental notions of computation, identifying computability with the computation of functions, as well as with TMs.
Claim 2. All computable problems can be described by an algorithm.
Reason: Adoption of algorithms as the central and complete concept of computer science.
Claim 3. Algorithms are what computers do.
Reason: Broadening the concept of an algorithm to make it more practical.

We will investigate these claims and analyze why they have emerged in the computer science community.

3 What is Computation?

The first reason for the Turing Thesis myth is related to the basic understanding of the notion of computation, as adopted by the theory of computation community; we refer to it as the "mathematical worldview".

3.1 The mathematical worldview

The theory of computation predates the establishment of computer science as a discipline, having been a part of mathematics before the 1960's. Its founders include such notable mathematicians as Godel, Kleene, Church, and Turing.[3] Mathematicians naturally equated the notion of computability with the computation of functions. Martin Davis's 1958 textbook [Dav58], popular among computer scientists, reflected the *mathematical worldview* that all computation is function-based, and therefore captured by TMs. It begins as follows:

[3] 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.

> "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, Rabin, and Scott. 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 1950's and 1960's. The mathematical worldview is the first of the three claims that constitute the Turing Thesis myth.

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

Turing thesis corollary: A problem is *solvable* if there exists a Turing machine for computing it.

The legitimacy of this corollary is based on two premises. The first one is the 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 perceived validity of this corollary was greatly 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 6.1.

3.2 The interactive paradigm

The mathematical worldview can be contrasted with the *engineering worldview*, where computation is viewed as an ongoing transformation of inputs to outputs – e.g., control systems, or operating systems. The question "*what do operating systems compute?*" has been a conundrum for the theoretical community, since they 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* [Weg97]:

Driving home from work: create a car that is capable of driving us home from work, where the locations of both work and home are provided as input parameters.

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 would have to be detailed enough so the car could predict the direction and strength of the wind at each point in 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 [EGW04], this is impossible – there is no such computable function. However, the problem *is* computable by a control mechanism, as in a robotic car, that continuously receives video input of the road and actuates the wheel and brakes accordingly.

The computation performed by automatic cars and operating systems is *interactive*, where input and output happen *during* the computation, not before or after it. This approach, distinct from either concurrency theory or the theory of computation, 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; later inputs to the computation may 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 controlling factory equipment, are all legitimate problems on par with finding common factors or choosing the next move on a given chess board.

4 Algorithms and Computability

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 has been enthusiastically adopted by the computer science community, who have since broadened its meaning. This adoption and the subsequent broadening are sources of the second and third reasons for the Turing Thesis myth.

4.1 The original role of algorithms

Algorithms are "recipes" for carrying out function-based computation, that can be followed mechanically.

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*.

Knuth's famous and influential textbook, *The Art of Computer Programming, Vol. 1* [Knu68] in the late 1960's popularized the centrality of algorithms in computer science. In his discussion of an algorithm, 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 [Knu68]: "toss lightly until the mixture is crumbly." This problem is not algorithmic because it is impossible for a computer to know how long to mix; this may depend on 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 also illustrates the sort of problems that Knuth meant to exclude.

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 50's and refined through the 1960's, 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. Not surprisingly, this absence hindered the adoption of ALGOL by the industry for commercial applications.

Knuth's careful discussion of algorithmic computation remains definitive to this day; in particular, it serves as the basis of the authors' understanding of this term. His discussion of algorithms ensures their function-based behavior and guarantees their equivalence with TMs [Knu68]:

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

4.2 Algorithms made central

The 1960's saw a proliferation of undergraduate computer science (CS) programs; 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 [ACM65], which served as a basis of its 1968 recommendations for undergraduate CS programs [ACM69]; one of the authors (PW) was among the primary writers of the 1968 report.

ACM's description of CS [ACM65] 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, modeled by TMs, 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 can be found in all modern textbooks on algorithms or computability. This has proved tremendously convenient for computer scientists, by allowing us to describe function-based computation informally using "pseudocode", with the knowledge that an equivalent Turing machine can be constructed.

However, neither mathematics nor the ACM provided an explicit agreed-upon definition of an algorithm. As we will see, the inconsistencies in the various definitions of this term greatly contributed to the Turing Thesis myth.

4.3 Algorithms redefined

The 1960's decision by theorists and educators to place algorithms at the center of CS was clearly reflected in early undergraduate textbooks. However, various textbooks chose to define this term differently. While some textbooks such as [Knu68] were careful to explicitly restrict algorithms to those that compute functions, and are therefore TM-equivalent, most left the restriction unstated.

An early example is [HU69], one of the first textbooks on the theory of computation (whose later editions are being used to this day). Their discussion of algorithms does not *explicitly* preclude non-functional computation, such as driving home from work:

> "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."

However, the prohibition against obtaining inputs dynamically during the computation is *implicitly* present in [HU69]. After all, ALGOL, the language then used for writing algorithmic programs, did not offer any constructs for input and output. Their examples of various problems also make it clear that only function-based computation was considered.

Yet other early textbooks, such as [Rice69], explicitly broadened the notion of algorithms to include problems beyond those that can be solved by TMs. On the surface, their definition of an algorithm is no different from [HU69]:

> "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.

The subject of [Rice69] 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. This approach, reflecting the centrality of algorithms without being restricted to the computation of functions, is typical of non-theory textbooks.

[Rice69] made no claims of TM-equivalence for their "algorithms". However, the students were not made aware that their notion of algorithms is different from Knuth's, and that the set of problems considered computable had thereby been enlarged. By pairing Rice's broader 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 impression that this broader set of problems could also be solved by TMs, giving rise to the Turing Thesis myth.[4]

4.4 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]. While the practical computer scientists have long since followed the lead of [Rice69] 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 [Sip97]. 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 their traditional selection of computational problems is all function-based, this description 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.

The result of this dichotomy is the Strong Turing Thesis. It is commonplace in the computing literature, including [Sip97]:

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

5 What is a Turing Machine?

Other claims have been used in support of the Strong Turing Thesis. We do not believe they played a major role in giving rise to the myth, yet they continue to serve as "corroboration" of its correctness. In this section, we discuss two such claims:

Claim 4. TMs serve as a general model for computers.
Reason: Misunderstanding the definition of a TM.

Claim 5. TMs can simulate any computer.
Reason: Misattributing general purpose computing to TMs.

These claims, related to the limitations of early computers, are discussed in this section.

[4] In private conversation with one of the authors (DG) in the fall of 1999, Knuth expressed some misgivings about his definition of an algorithm, and shared plans to broaden it if that text were ever rewritten. It is not clear what his plans were regarding the claim of equivalence between algorithms and TMs.

5.1 Syntax and semantics

According to the set-theoretic definition, 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 [Weg68], 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:

Nature of computers: TMs serve as a general model for computers.

This claim has been used in support of the Turing Thesis myth.

Just as for any other class of automata such as FSA (finite state automata), the set-theoretic definition does not completely capture TMs; it captures their *syntax*, but not *semantics*.

Syntax of a TM: what does it consist of?
Semantics: how does it compute?

As defined by Turing [Tur36], 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 alternative models of *choice machines* and *oracle machines*. The complete TM definition includes both the set-theoretic syntactic definition and the semantic definition.

> Statements about TM expressiveness, such as the Turing Thesis, fundamentally depend on their semantics, as defined by Turing. If these semantics were defined differently, it may (or may not) produce an equivalent machine.

Early computers did in fact compute as prescribed above. However, while perhaps reflecting TM syntax, 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); 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; 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* computation.

5.2 The Universal Turing Machine

A *Universal Turing Machine* is a special TM introduced by Turing [Tur36], that can simulate any other TM. It 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. As long as each machine in that class can be captured by a finite description, prescribing what this machine would do in every possible configuration, a TM can be created to simulate all machines in that class:

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

Analogously to the Turing Thesis, the Universality Thesis combines with the mathematical worldview to obtain the following corollary:

Universality Corollary: Any computer can be simulated by a TM.

This corollary is the second of the two claims that have been used to "corroborate" the Turing Thesis myth. Again, the undergraduate textbooks played a key role. In order to present the expressiveness of TMs in a more accessible (and dramatic) fashion, the Universality Corollary was melded with the Turing Thesis Corollary, resulting in the following statement that (incorrectly) summarized the role of TMs:

Strong Turing Thesis: A TM can do anything that a computer can do.

6 Time for New Models

6.1 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 [Tur36], and all the versions of these machines that are equivalent to the original. Indeed, 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. In general, the equivalence of TM versions cannot be taken for granted. For example, if only right transitions are allowed, the resulting model is not equivalent, having the expressiveness of an FSA rather than a TM.

The example above is 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 alhabet, are algorithmic. In the case of algorithmic extensions, the Church-Turing thesis applies, and it can be taken from granted that the new model is equivalent to the original. However, as a result of the Turing Thesis myth, it is common to assume the equivalence of any TM extension to the original, and we no longer 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 outpus 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". Indeed, the recent proliferation of such models can be viewed as a paradigm shift in the field of models of compuation.

While they usually share TM syntax, the semantics of these new extensions is different; for example, in the case of persistent Turing machines, it is based on dynamic input streams and persistence. Out of habit, researchers have continued to assume that these extensions are equivalent to the original TM. But in the case of such non-algorithmic extensions, Turing's thesis does not apply, and equivalence can no longer be taken for granted. Indeed, it no longer holds, as discussed next.

6.2 Modeling interactive computation

Wegner [Weg97,Weg98] has conjectured that interactive models of computation are more expressive than "algorithmic" ones such as Turing machines. It would therefore be interesting to see what minimal extensions are necessary to Turing machines to capture the salient aspects of interactive computing. Motivated by these goals, [GSAS04] investigates a new way of interpreting Turing-machine computation, one that is both interactive and persistent – *persistent Turing machines* (PTMs); we discuss this work here.

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. Upon receiving an input token from its environment on its input tape, a PTM computes for a while and then outputs the result to the environment on its output tape, and this process is repeated forever. A PTM computes *concurrently* with its *environment*, both acting as consumers of each other's outputs and producers of each other's inputs.

In addition to having *dynamic stream semantics*, PTM computations are *persistent* in the sense that a notion of "memory" (work-tape contents) is maintained from one computation step to the next, where each PTM computation step represents an N3TM computation. *Decider PTMs* are an important subclass of PTMs; a PTM is a *decider* if it does not have divergent (non-halting) computations.

The notions of interaction and persistence in PTMs are formalized in terms of the *persistent stream language* (PSL) of a PTM. Given a PTM, its persistent stream language is the set of infinite sequences (interaction streams) of pairs of the form (w_i, w_o) representing the input and output strings of a single PTM computation step. Persistent stream languages induce a natural, stream-based notion of equivalence for PTMs.

The first result concerning PTMs is that the class of PTMs is isomorphic to *interactive transition systems* (ITS), a very general kind of effective transition systems, thereby allowing one to view PTMs as ITSs "in disguise". ITSs come with three notions of behavioral equivalence: *ITS isomorphism*, *interactive*

bisimulation, and *interactive stream equivalence*, where ITS isomorphism refines interactive bisimulation, and interactive bisimulation refines interactive stream equivalence.

A similar result is established for decider PTMs and decider ITSs. These results address a question heretofore left unanswered concerning the relative expressive power of Turing machines and transition systems, namely: "What extensions are required of Turing machines so that they can simulate transitions systems?"

An infinite hierarchy is defined, of successively finer equivalences for PTMs over finite interaction-stream prefixes; it is shown that the limit of this hierarchy does *not* coincide with PSL-equivalence. The presence of this "gap" can be attributed to the fact that the transition systems corresponding to PTM computations naturally exhibit *unbounded nondeterminism*. In contrast, classical Turing-machine computations have bounded nondeterminism, i.e., any nondeterministic TM can produce only a finite number of distinct outputs for a given input string.

Amnesic PTMs are a special type of PTMs where each new computation begins with a blank work tape, with a corresponding notion of *amnesic stream languages* (ASLs). The class of ASLs is strictly contained in the class of PSLs. Also, ASL-equivalence coincides with the equivalence induced by considering interaction-stream prefixes of length one, the bottom of the equivalence hierarchy; this hierarchy therefore collapses in the case of amnesic PTMs.

ASLs are representative of the classical view of Turing-machine computation, extending TMs with dynamic stream-based semantics but without persistence. One may consequently conclude that, in a stream-based setting, the extension of the Turing-machine model with persistence is a nontrivial one.

Finally, the notion of a *universal PTM* is introduced. Similarly to a *universal Turing machine*, a universal PTM can simulate the behavior of an arbitrary PTM. The class of *sequential interactive computations* is also introduced:

Sequential Interactive Computation: A sequential interactive computation continuously interacts with its environment by alternately accepting an input string and computing a 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 interaction include sequential JAVA objects, static C routines, single-user databases, network protocols, and our original example of driving home from work. A sequential interactive analogue to the Turing Thesis is provided:

Sequential Interaction Thesis: Any sequential interactive computation can be performed by a persistent Turing machine.

This hypothesis, when combined with other results in the paper, implies that the class of sequential interactive computations is more expressive than the class of algorithmic computations, and thus is capable of solving a wider range of problems – proving Wegner's conjecture.

TMs capture *effective* transformations over finite strings, but the notion of effectiveness also applies to operations on higher-level objects such as functions, see the theory of *recursive functionals* [Rog67,San92]. Extensions are needed to the Church-Turing thesis to capture this effectiveness [San92]; it is conjectured that the Sequential Interaction Thesis captures the expressiveness of recursive functionals.

It has been also conjectured [Weg98] that *multi-agent* interaction is more expressive than sequential, or *single-agent* interaction. These conjectures remain to be proven.

6.3 Turing Thesis myth corrected

We have discussed the origins for the popularity of the Turing Thesis myth, having identified three distinct claims that comprise it:

Claim 1. (Mathematical worldview) All computable problems are function-based.
Claim 2. (Focus on algorithms) All computable problems can be described by an algorithm.
Claim 3. (Practical approach) Algorithms are what computers do.

Furthermore, we looked at two more claims that have been used to corroborate the Turing Thesis myth:

Claim 4. (Nature of computers) TMs serve as a general model for computers.
Claim 5. (Universality corollary) TMs can simulate any computer.

For each of these claims, there is 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:

Corrected Claim 1. All algorithmic problems are function-based.
Corrected Claim 2. All function-based problems can be described by an algorithm.
Corrected Claim 3. Algorithms are what early computers used to do.
Corrected Claim 4. TMs serve as a general model for early computers.
Corrected 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 claim, while incompatible with original claims, is perfectly consistent with their corrected versions. It contradicts the Strong Turing Thesis, exposing the fallacy of the Turing Thesis myth. Dispelling this myth has grown more important as the practice of computing is becoming more and more interactive. Its algorithmic fundamental identity no longer serves it well.

7 Conclusion

Hoare, Milner and others have long realized that TMs do not model all of computation [WG03]. However, when their theory of concurrent computation was first developed in the late '70s, it was premature to openly challenge TMs as a complete model of computation. Concurrency 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.

Researchers in other areas of theoretical computer science have also found need for interactive models of computation, such as Input/Output automata for distributed algorithms [LT89] and Interactive TMs for interactive proofs [GMR89]. However, the issue of the expressiveness of interactive models vis-a-vis TMs was not raised until the mid-1990's, when the model of interaction machines as a more expressive extension of TMs was first proposed by one of the authors [Weg97].

The theoretical framework for sequential interaction machines, as a persistent stream-based extension to TMs, was completed by the other author in [GSAS04]; it is discussed above. Van Leeuwen, a Dutch expert on the theory of computation, proposed an alternate extension in [VLW00]. In addition to interaction, other ways to extend computation beyond Turing machines have been considered, such as quantum computing.

While not part of CS Theory, the field of AI has perhaps gone the furthest in explicitly recognizing the expressiveness gains of moving beyond algorithms. In the early 1990's, Rodney Brooks convincingly argued against the algorithmic approach of "good old-fashioned AI", positioning interaction is a prerequisite for intelligent system behavior [Bro91]. This argument has been adopted by the mainstream AI community, whose leading textbooks recognize that interactive *agents* are a better model of intelligent behaviors than simple input/output functions [RN94].

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. A paradigm shift is necessary in our notion of computational problem solving so it can better model the services provided by today's computing technology.

The *Strong Church-Turing Thesis* reinterprets the Church-Turing Thesis to imply that Turing Machines model all computation. In this paper, we provided a new analysis of the historical reasons for the widespread acceptance of the myth that the two versions of the thesis are equivalent. However, the assumption that all of computation can be algorithmically specified is still widely accepted in the CS community, and interaction machines have been criticized as an unnecessary Kuhnian paradigm shift.

By facing the fact that this reinterpretation is a myth we can move forward with formal models of interaction machines, which extend Turing machines with interaction to capture the behavior of concurrent systems. We presented one such model, *Persistent Turing Machines* (PTMs), that promises to bridge theory of computation and concurrency theory. PTMs capture 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 are have been shown to be more expressive than Turing machines. It is hoped that PTMs will lay the foundation for a new theory of interactive computation which will bridge the current theories of computation and concurrency.

Acknowledgements

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

References

[ACM65] An Undergraduate Program in Computer Science-Preliminary Recommendations, A Report from the ACM Curriculum Committee on Computer Science. *Comm. ACM* 8(9), Sep. 1965, pp. 543-552.

[ACM69] Curriculum 68: Recommendations for Academic Programs in Computer Science, A Report of the ACM Curriculum Committee on Computer Science. *Comm. ACM* 11(3), Mar. 1968, pp. 151-197.

[Bro91] R. Brooks. Intelligence Without Reason. *MIT AI Lab Technical Report 1293.*

[Dav58] M. Davis. *Computability & Unsolvability.* McGraw-Hill, 1958.

[Den04] P. Denning. The Field of Programmers Myth. *Comm. ACM*, July 2004.

[EGW04] E. Eberbach, D. Goldin, P. Wegner. Turing's Ideas and Models of Computation. In *Alan Turing: Life and Legacy of a Great Thinker*, ed. Christof Teuscher, Springer 2004.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comp.*, 18(1):186–208, 1989.

[GSAS04] D. Goldin, S. Smolka, P. Attie, E. Sonderegger. Turing Machines, Transition Systems, and Interaction. *Information & Computation J.*, Nov. 2004.

[HU69] J.E. Hopcroft, J.D. Ullman. *Formal languages and their relation to automata.* Addison-Wesley, 1969.

[Knu68] D. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, 1968.

[LT89] N. Lynch, M. Tuttle. An Introduction to Input/Output automata. *CWI Quarterly*, 2(3):219–246, Sep. 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[Rice69] J. K. Rice, J. N. Rice. *Computer Science: Problems, Algorithms, Languages, Information and Computers.* Holt, Rinehart and Winston, 1969.

[RN94] S. Russell, P. Norveig. *Artificial Intelligence: A Modern Approach.* Addison-Wesley, 1994.

[Rog67] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[San92] L. Sanchis. *Recursive Functionals*, North Holland, 1992.

[SIGACT] *SIGACT News*, ACM Press, March 2004, p. 49.

[Sip97] M. Sipser. *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

[Tur36] A. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem, *Proc. London Math. Soc.*, 42:2, 1936, pp. 230-265; A correction, ibid, 43, 1937, pp. 544-546.

[VLW00] J. v. Leeuwen, J. Wiedermann. The Turing Machine Paradigm in Contemporary Computing. in *Mathematics Unlimited - 2001 and Beyond*, eds. B. Enquist and W. Schmidt, Springer-Verlag, 2000.

[Weg68] P. Wegner. *Programming Languages, Information Structures and Machine Organization*, McGraw-Hill, 1968.

[Weg97] P. Wegner. Why Interaction is More Powerful Than Algorithms. *Comm. ACM*, May 1997.

[Weg98] P. Wegner. Interactive Foundations of Computing. *Theoretical Computer Science* 192, Feb. 1998.

[WG03] P. Wegner, D. Goldin. Computation Beyond Turing Machines. *Comm. ACM*, Apr. 2003.