Behavior and Expressiveness of Persistent Turing Machines

Peter Wegner and Dina Goldin

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-99-14
September 1999

Behavior and Expressiveness of Persistent Turing Machines

Dina Goldin, University of Massachusetts at Boston <dqg@cs.umb.edu>
Peter Wegner, Brown University <pw@cs.brown.edu>

Abstract. *Persistent Turing machines* (PTMs) are multitape machines with a persistent worktape preserved between successive interactions. They are a minimal extension of Turing machines (TMs) that express interactive behavior. Their behavior is characterized by input-output streams; PTM equivalence and expressiveness are defined relative to its behavior. Four different models of PTM behavior are examined: *language-based*, *automaton-based*, *observation-based*, and *function-based*. A number of special subclasses of PTMs are identified, and several expressiveness results are obtained, both for the general class of all PTMs and for the special subclasses. The methods, formalisms, and tools for studying models of PTM computation developed in this paper can serve as a basis for a more comprehensive theory of interactive computation.

1. Introduction

Persistent Turing Machines (PTMs) are multitape machines with a persistent worktape preserved between successive interactions; they are a minimal extension of Turing machines (TMs) that express interactive behavior [GW]. They model services over time provided by embedded, object-oriented, or reactive systems that cannot be expressed by computable functions [MP, We2, WG].

TMs and PTMs are abstract computing devices useful for representing different forms of computational behavior: TMs model *algorithmic* behavior whereas PTMs model *sequential interactive behavior* (SIMs) [WG]. In general, sequential interactive behavior may involve aspects that are not modeled by PTMs, with temporal or non-discrete characteristics; these characteristics may be crucial, as in the case of embedded devices.

PTMs share with TMs the inability to express temporal or non-discrete characteristics of computation. Nevertheless, both TMs and PTMs are robust computational abstractions which serve as a foundation for formal study of algorithmic and sequential interactive computation, respectively. We believe that the PTM model can be extended to account for non-functional behavioral characteristics of computing devices, by augmenting models of behavior beyond streams of input/output pairs.

PTM equivalence and expressiveness are defined relative to its behavior. Models of computation for PTMs, just as models of computation for TMs, can be equivalently expressed by languages, automata, observation-based models, and function-based models. Whereas models of computation for TMs are string-based, models for PTMs are based on interaction (input/output) streams. We use these classes of models to obtain several expressiveness results, both for the general class of PTMs and for a number of special subclasses.

Summary: Section 2 provides some background concepts and defines PTMs. Section 3 outlines the different models of PTM behavior, identifies several PTM subclasses, and summarizes the expressiveness results about them. Sections 4 through 7 give the details of the four models and obtain the corresponding expressiveness results, followed by a conclusion.

2. Extending Turing Machines with Persistence

2.1 Turing Machines as Noninteractive Computing Devices

Turing machines are finite agents that transform input into output strings by sequences of state transitions.
Turing machine (TM): a TM is a state-transition machine $M = (S, \Sigma, \delta)$, with finite sets of states S and

tape symbols Σ, and a state-transition relation δ: SxΣ -> Sx{Σ,L,R}. TMs transform finite input strings x ∈ Σ* to outputs y = *M*(x) by a finite sequence of steps, staring in a unique *starting* state and ending when a *halting* state is reached. At each step, *M* reads a tape symbol i, performs a state transition (s,i) -> (s',o), writes a symbol o, and/or moves the reading head one position right or left [HU]. ■

TMs compute functions f: X->Y from integers to integers (strings to strings). The class of functions computable by TMs are called the *computable function*s. TM computations for a given input are history-independent and reproducible, since TMs always start in the same initial state and obtain all their input prior to start of computation (they "shut out the world" during computation).

The property of "shutting out the world" during a computation, where all input is determined prior to the start of the computation, characterizes Turing Machines as *non-interactive*:
Interactive computing devices: *Interactive computing devices* admit input/output actions during the process of computation [We2]. ■

Several classic extensions to the TM model have been proposed, such as multiple tapes or multiple read/write heads. A *multitape* TM typically has 3 or more tapes: a read-only input tape, one or more internal work tapes, and a write-only output tape. This extension, and others proposed, are noninteractive; they have been shown to have no effect on the expressiveness of the TM model [HU].

As we will show in this paper, when inputs are streams with dynamic binding, interactive computing devices can be more expressive than TMs.

2.2. Persistent Turing Machines

Persistence of state typifies computing devices as diverse as software objects and intelligent agents [AB, RN]. In this section, we extend Turing Machines by introducing persistence. It is interesting to note that Turing foresaw interactive Turing Machines in his seminal paper [Tu], where he made a distinction between *automatic machines*, now known as TMs, and *choice machines*, which allow choices by an external operator.

Persistent Turing machines (PTMs) are a minimal extension of TMs [GW] that models interactive computation:
Persistent Turing Machine (PTM): A PTM is a multitape TM with a *persistent work tape* whose content is preserved between successive TM computations, known as PTM *computation steps*. ■

Since the work tape at the beginning of a PTM computation step is not necessarily blank, the output of a PTM *M* at the end of the computation step depends both on the on the input and on the work tape contents. As a result, *M* defines a *partial recursive function* f_M from (input, worktape) pairs to (output, worktape') pairs of strings: f_M: I x W -> O x W

Example 2.2 (Answering Machine): An *answering machine A* is a PTM whose worktape contains a sequence of recorded messages and whose operations are "record message", "playback", and "erase". The Turing-computable function for *A* is:
f_A(record Y, X) = (ok, XY); f_A(playback, X) = (X, X); f_A(erase, X) = (done, ε)
Note that both the content of the worktape and the length of input for recorded messages are unbounded. ■

A PTM's persistent works tape contents is known as its *state*:
PTM state: A PTM's persistent work tape is known as the its *memory.* The contents of the memory prior before and after a (single TM) computation is known as the PTM's *state*. ■

PTM states can be represented by strings, so the set of PTM states is enumerably infinite. In contrast, TMs have a finite set of states.

2.3 PTM Computations

Individual computation steps of a PTM correspond to TM computations and are string-based. However, the semantics of PTM computations are *stream-based*; the input stream is generated by the *environment* (*observer*) and the output stream is generated by the PTM (the computing device).
Stream-based PTM semantics: both the input and the output of a PTM are streams [BM] whose tokens are strings over Σ^***.** These streams are sequences of tokens with *dynamic (late, lazy*) evaluation semantics, where the next value is not generated until the previous one is consumed.■

A *PTM computation* is an (infinite) sequence of Turing-computable steps, consuming input stream tokens and generating output stream tokens:
PTM computation: Given a PTM M defining a function f_M and an input stream (i_1, i_2, ...), a computation of M consists of a sequence of computational steps and produces an output stream (o_1, o_2, ...):

$$f_M(i_1, w_0) = (o_1, w_1);\ f_M(i_2, w_1) = (o_2, w_2);\ f_M(i_3, w_2) = (o_3, w_3);\ \ ...$$

The state of the PTM *evolves* during a PTM computation, starting with the initial state w_0: (w_0, w_1, w_2, ...); w.l.o.g., we can assume that w_0 is an empty string. ■

The fact that input and output actions on streams take place in the middle of computation characterizes PTMs as interactive computing devices. The evolution of the PTM state is in contrast to the TMs, which always start (end) a computation in the same starting (halting) state (section 2.1).

Example 2.3 (Answering machine computation): For the Answering Machine (example 2.2), the input stream (record A, erase, record BC, record D, playback...) generates the output stream (ok, done, ok, ok, BCD, ...); the state evolves as follows: (ε, A, ε, BC, BCD, BCD, ...). ■

From this example, it is easy to see how any abstract data type can be implemented as a PTM. Other examples of interactive behaviors that can be expressed by PTMs include single-user databases, holding a dialogue, driving home from work [We1] and playing two-person games [Ab].

2.4 PTM interaction streams

The interaction of PTMs with their environment can be described by *interaction streams*, which interleave PTM's inputs and outputs:
Interaction (I/O) streams have the form (i_1,o_1), (i_2,o_2), ... , where i's are input tokens generated by the environment (observer) and o's are output tokens generated by the PTM (computing device). For all k, o_k is computed from i_k but precedes and can influence i_{k+1}. ■

Due to the dynamic evaluation semantics of PTM input streams, it is assumed that each input token is generated after the previous output. As a result, later input tokens may depend on earlier outputs and on external (exogenous) events in the environment.

Example 2.4 (Interaction stream for the Answering Machine): the computation in example 2.3 leads to the following interaction stream:
((record A, ok), (erase, done), (record BC, ok), (record D, ok), (playback, BCD), ...). ■

Interaction streams are *history dependent*. This can be seen in the example above, where the fifth output BCD depends on the third and fourth inputs. The history dependent effect of "playback" cannot be expressed by TMs whose output is a function of the input.

3. Towards formalizing PTMs

Being able to determine when two computing devices are equivalent, or when one class of devices is more expressive than another, is an important goal for any attempt to a formalization of a new model for computing devices. We will formulate these notions in terms of a computing device's behavior, and formalize them for PTMs by providing specific models of PTM behavior.

3.1 Behaviors, equivalence, and expressiveness

The notions of equivalence and expressiveness of computing devices are intimately related to the notion of a device's *behavior*, where behavior is exhibited through its interactions with the environment.

Example (TM behavior): The behavior of a TM consists of accepting input strings and returning output strings; this can be alternately modeled as a set of input/output pairs or as a function from inputs to outputs. ■

PTM interaction streams (section 2.4) completely capture the behavior of PTMs, though it can be modeled in different ways, just as for TMs.

The notion of *equivalence* applies to individual pairs of devices, by comparing their behaviors:
Definition 3.1.1 (equivalence): Two computing devices are *equivalent* if they have the same behavior. ■

The notion of *expressiveness* applies to classes of computing devices, by comparing the sets of behaviors for them:
Definition 3.1.2 (expressiveness): Two classes of computing devices C1 and C2 are said to have the same *expressiveness* iff the two corresponding sets of behaviors are equal; C2 is said to be *more expressive* than C1 if the set for C1 is a proper subset of that for C2. ■

Example (FSAs and PDAs): for every Finite State Automaton (FSA), there exists an equivalent Push-Down Automaton (PDA), one which accepts the same language. However, the reverse is not true; as a result, PDAs are more expressive than FSAs [HU]. ■

In this paper, we show that PTMs have a richer set of behaviors than TMs, and are therefore more expressive than TMs in precisely the same sense that PDAs are more expressive FSAs.

3.2 Models of PTM behavior

Four different models of PTM behavior are examined: *language-based*, *automaton-based*, *observation-based*, and *function-based*. Whereas models of computation for TMs are string-based, models for PTMs are based on interaction streams (section 2.4):

Language-based: A PTM M is modeled by its *language*, which is the set of all interaction streams for M (section 4).

Automaton-based: A PTM M can be modeled by an infinite-state transducer whose transitions are labeled by pairs of input/output strings; paths through that automaton are the interaction streams of M (section 5).

Observation-based: A PTM M can be modeled by a set of finite *observations*, made by M's *observer.* These observations are segments of interaction streams, giving us notions of behavior and equivalence that are relative to observers (section 6).

Function-based: A PTM can be modeled by a recursive function from input to output streams; this function has coinductive evaluation semantics [BM], recursing infinitely without reaching a base case (section 7).

As for Turing Machines, different models of computation for PTMs are suitable for obtaining different results. In the next section, we define special specializations of PTMs as a prelude to proving results about their expressiveness.

3.3 Special PTMs and their expressiveness

In general, a single computational step of a PTM *M* is modeled by a Turing computable function f_M: I x W -> O x W where I, O, and W are Σ* (section 3.1). Several interesting subclasses of PTMs are defined in this section, which arise from subcases of this general model:

Amnesic PTMs (section 4.2) ignore the contents of their memory, behaving as if each computational step starts in the same state.

Finite-memory PTMs (section 4.3) have a bound on the amount of available memory.

Finite-state PTMs (section 4.3) have a finite number of states; that is to say, W is finite.

Simple PTMs have finite sets of input/output tokens; that is to say, both I and O are finite; **binary PTMs** are a subclass of simple PTMs where both I and O have size 2.

Consistent PTMs produce the same output token for a given input token everywhere within a single interaction stream; different interaction streams may have different outputs for the same input [Kos].

These subclasses of devices have special characteristics. Here are some of them:

* Amnesic PTMs have the same expressiveness as sequences of TM computations (section 4);
* Finite-memory PTMs have the same expressiveness as finite-state PTMs (section 4);
* Simple finite-state PTMs have the same expressiveness as Finite State Transducers (section 5);
* Simple (binary) finite-state PTMs are less expressive than simple (binary) PTMs (section 5);
* General PTMs have the same expressiveness as *effective labeled transition systems* (LTSs), a subclass of LTSs (section 5);
* Amnesic PTMs are less expressive than general PTMs, constituting the lowest level of an *infinite observability hierarchy* for PTMs (section 6).

The remainder of the paper, up to the concluding section, is devoted to a discussion of the above results.

4. Language-based model of PTM computation

In this section, we use the language-based model of PTM computation to show that the class of *amnesic* PTMs (section 3.3) has the same expressiveness as the class of TMs, and that the class of *finite-memory* PTMs has the same expressiveness as the class of *finite-state* PTMs. Languages and grammars for interaction machines were previously explored in [We2].

4.1 PTM languages

The behavior of non-interactive computing devices can be described by a *language*, which is a set of strings over some finite alphabet. For example, the language of a finite state automaton is the set of strings it accepts [HU]. A set-based notion of languages can likewise be defined for PTMs; PTM languages are stream-based rather than string-based:
PTM language: The set of all interaction streams for a PTM *M* constitutes its *language*, $\mathcal{L}(M)$. ■

Despite the change in semantics from strings to streams, the definitions of expressiveness and equivalence (section 3.1) apply to PTMs just as to other computing devices for which a language is defined:

Language-based equivalence: Let M1 and M2 be computing devices whose behavior is modeled by sets of interaction streams; M1 and M2 are *equivalent* iff $\mathcal{L}(M1) = \mathcal{L}(M2)$.

Language-based expressiveness: Let C1 and C2 be two classes of computing devices whose behavior is modeled by sets of interaction streams; C2 is said to be *more expressive* than C1 if the set of all languages for the members of C1 is a proper subset of that for C2. ■

4.2 Amnesic PTMs and TMs

In order to compare PTM languages to those of multitape TMs, we need to consider TM computations over streams of input strings, which produce streams of output strings:
Stream-based TM computation: Given a TM M defining a function f_M from input to output strings and an input stream $(i_1, i_2, ...)$, a computation of M produces an output stream $(o_1, o_2, ...)$, where $f_M(i_k) = (o_k)$ for each k. ■

Amnesic PTMs ignore the contents of their memory, behaving as if each computational step starts in the same state. As a result, computations of amnesic PTMs are not history dependent (hence their name):
Definition 4.2 (Amnesic PTMs): a PTM M is *amnesic* iff $f_M(i, w) = f_M(i, w_0)$ for all inputs i and states w, where w_0 is M's initial state. ■

Proposition 4.2.1: the behavior of each amnesic PTM is equivalent to a sequence of TM computations for the corresponding TM.
Proof: Given an input stream $(i_1, i_2, ...)$, a computation of an amnesic PTM M produces an output stream $(o_1, o_2, ...)$, where:

$$f_M(i_1, w_0) = (o_1, w_1);\ f_M(i_2, w_1) = f_M(i_2, w_0) = (o_2, w_2);\ f_M(i_3, w_2) = f_M(i_2, w_0) = (o_3, w_3);\ ...$$

Since w_0 is assumed empty, each computation step of M is the same as a computation of the TM that corresponds to M, let us call it *M0*: for all k, $f_{M0}(i_k) = f_M(i_k, w_0) = o_k$. ■

Proposition 4.2.2: for each TM M, there exists a TM M' so that Ms behavior is equivalent to the amnesic PTM based on M'.
Proof: Given an arbitrary TM M, it may write something to tape and fail to erase it, so that the corresponding PTM is not necessarily amnesic. However, an equivalent TM M' that always erases its work tape can be constructed, guaranteeing that the behavior of the corresponding PTM is amnesic. ■

The following theorem follows from Propositions 4.2.1 and 4.2.2:
Theorem 4.2: Amnesic PTMs have the same expressiveness as sequences of TM computations. ■

4.3 Finite-state and finite-memory PTMs

Finite-memory PTMs have a bound on the amount of available memory. This does not imply a bound on the size of the work tape, but only on the number of bits that may be stored persistently.
Let M be a finite-memory PTM. If n is the memory bound for M, and c is the size of M's alphabet Σ, then M can have at most cn states. Therefore, the set of states for M is finite.

Finite-state PTMs have a finite number of states; that is to say, given the set of all possible finite sequences of inputs, the set of states that can be reached by these sequences is finite.

Let M be a finite-state PTM. If W is the set of M's states and k is the size of W, then at most log k charac-

ters are needed to represent any state in W. Therefore, one can place a bound on the amount of memory available to *M* without affecting its computation.

The following theorem follows from the discussion above:
Theorem 4.3: Finite-memory PTMs have the same expressiveness as finite-state PTMs. ■

5. Automata-theoretic model of PTM behaviors

In this section, we adopt the automata-theoretic view of a PTM. as a transducer over an infinite state space. This view complements the language-based view, just as for TMs.

5.1 Finite-state transducers and their index

One of the possible metrics for classify finite-state automata (FSAs) is their *index*, which is finite for all FSAs and is the same for equivalent FSAs [RS]:
Myhill-Nerode theorem: Given any finite state automaton *A* over alphabet Σ, where L(*A*) is the set of strings accepted by *A*, it induces a partition of Σ* into $\{L_1, L_2 \ldots L_k\}$ for some finite *k*, where:

for any $i \le k$, for all x, y in L_i, and for all w in Σ*, xw is in L(*A*) iff yw is in L(*A*);

k is known as the *index* of A. ■

Finite state transducers (FSTs) augment FSAs with character outputs at each transition (rather than the binary output *accept*/*reject* at the end of the computation). A Mealy machine is an example of an FST:
Mealy Machine: a Mealy machine (Q, Σ, Δ, δ, λ, q0) is a finite state machine where each transition is associated with both an input and an output character. For a transition from state *q* in Q with input label *a* in Σ, the value of the next state is δ(*q*, *a*); the value of the output is λ(*q*, *a*), which is a member of Δ. [HU] ■

Moore machines, where the output is associated with a state rather than with a transition [HU], are another example of a transducer; Moore and Mealy machines have the same expressiveness [HU].

For inputs of finite length, FSTs can simulate any FSA: restrict the output alphabet Δ to two values, *accept* and *reject*, and consider only the last character of the output string. However, the input of an FST may be an infinite character stream; the output will also be a character stream. Myhill-Nerode theorem can be adapted to stream-based FSTs [TB]:
Proposition 5.1 (Myhill-Nerode theorem for FSTs): Given any finite state transducer *A* over alphabets Σ and Δ, which maps streams over Σ to streams over Δ by a mapping μ, *A* induces a partition of Σ* into $\{L_1, L_2 \ldots L_k\}$ for some finite *k*, where:

for any $i \le k$, for all x, y in L_i, and for all w in Σ*, the last characters of μ(xw) and μ(yw) are the same;

k is known as the *index* of *A*. ■

5.2 FSTs and PTMs

Stream-based Mealy machines are the same as *finite-state simple* PTMs, which have finite sets of input/output tokens:
Proposition 5.2.1: Stream-based Mealy machines have the same expressiveness as *finite-state simple* PTMs.
Proof: given a finite-state simple PTM *M* with f_M: I x W -> O x W, it is easy to see how to construct an equivalent Mealy Machine (Q, Σ, Δ, δ, λ, q0) , and vice versa. ■

As a corollary to Proposition 5.2.1, finite-state simple PTMs have a finite index. In contrast, simple PTMs without the finite-state restriction may have no finite index:
Proposition 5.2.2: Myhill-Nerode theorem for FSTs does not hold in the case of infinite-state transducers:
Proof: Let *M* be the following PTM, which accepts and produces bit streams as follows:

while *M*'s input stream equals the "template" $\sigma = 0101^2 01^3 01^4 0\ldots$, *M*'s output bits are all 1's; as soon as there is an "incorrect" input bit, *M*'s output becomes 0's.

Let *L* be the set of all prefixes of the template σ that end in 0; *L* is countably infinite. Let $\mu(x)$ be the output string produced by *M* for any input string *x*. It is easy to see that given any two distinct strings x, y in *L*, there exists a string w of the form 1^n for some $n \geq 1$ such that the last character of μ(xw) is different from the last character of μ(yw). ■

As Propositions 5.1 and 5.2.2 show, the index of any finite-state simple PTM is finite, whereas the same is not true for the class of all simple PTMs. Therefore, the behavior of the class of simple PTMs can be considered richer than of the class of FSTs, when measured in terms of the index. Moreover, since PTM *M* used in the proof above is *binary* (section 3.3), the same difference in expressiveness is found for the class of binary PTMs.

5.3 LTSs and PTMs

There is a well-studied class of transition systems with infinite states that are known as Labeled Transition Systems [BM].
Definition (LTS): A *labeled transition system* is a triple (S, A, δ), where S is a set of *states*, A is a set of *actions*, and δ: S x A -> S is a *transition function*. ■

The transition function δ associates with every state a set of possible actions, and specifies the new state reached with each action. The actions are assumed to be observable whereas the states are not. LTSs are a standard tool for process modeling [Mil].

We identify a particular class of LTSs, called *effective* LTSs; each action of an effective LTS is an input/output pair, and the transitions are Turing-computable:
Effective LTSs: An LTS T is *effective* if A = I x O, where I is a set of *inputs* and O is a set of *outputs*, and the transition function δ is Turing-computable. ■

In fact, PTMs comprise a subclass of LTSs with trace equivalence semantics:
Proposition 5.3: The class of PTMs has the same expressiveness as the class of effective LTSs.
Proof: given a finite-state simple PTM *M* with f_M: I x W -> O x W, it is easy to see how to construct an equivalent effective LTS (S, I, O, δ), and vice versa. ■

6. Observational approach to PTM behaviors

System behavior has been defined as an absolute notion. However, *observed* behavior of systems defined relative to *observers* is in many contexts more relevant than unobservable absolute behavior. In this section, we consider an observational notion of behavior for interactive systems.

6.1 Observational equivalence of interactive systems

Observers map machines with behavior B to their *observed behaviors*:
Definition 6.1 (observational equivalence): Given a class C of machines with behavior B, an *observer* O of C is a mapping from elements of C to some domain β_O, O: C -> β_O which has the following property, known as *observational correctness*:

for any M_1, M_2 in C, if $B(M_1) = B(M_2)$, then $O(M_1) = O(M_2)$ (observational correctness)

The elements of β_O are known as *observed behaviors*. In case when $O(M_1) \neq O(M_2)$, we say that M_1 and M_2 are *distinguishable* to O. Otherwise, we say that M_1 and M_2 *are observationally equivalent* to O. ■

Observers should never distinguish between equivalent systems (as a result of *the observational correctness* property), but they may fail to distinguish between systems that are not equivalent. We will consider a special case of observers, known as *finite observers*:
Finite observers: O is a *finite observer* for C if, for all M in C, O(M) is a set of finite (bounded) elements, known as *observations*. ■

Finite PTM observers: For an arbitrary PTM *M*, any finite subsequence of some interaction stream of *M* is an *observation* of *M*; the *length* of an observation is the number of pairs in it. Then, any mapping from PTMs to sets of their observations constitutes a *finite PTM observer.* ■

If M_1 and M_2 are distinguishable to a finite observer O, then $O(M_1) \neq O(M_2)$, i.e., $O(M_1) \oplus O(M_2)$ is not empty. The members of this set are known as *distinguishability certificates* for (M_1, M_2); they are finite observations which allow O to distinguish between M_1 and M_2.

Proposition 6.1 (Distinguishability certificates for amnesic PTMs): if two amnesic PTMs are not equivalent, they have a distinguishability certificate of length 1.
Proof: Let M_1 and M_2 be two amnesic PTMs, and let O_1 be a finite PTM observer that observes all PTM intraction sequences of length 1; i.e., O_1 observes single input/output pairs . It is easy to see that M_1 and M_2 are distinguishable iff $O_1(M_1) \neq O_1(M_2)$.■

Proposition 6.1 does not hold for the general class of PTMs; in fact, we will show in section 6.3 that there exist pairs of PTMs whose shortest distinguishability certificate is arbitrarily long.

6.2 Observational expressiveness

It is easy to see that observational equivalence (definition 6.1) is symmetric, reflexive, and transitive. As a result, an observer O for a class of machines C with behavior B induces a partitioning of C into equivalence classes, where the members of each class are not distinguishable to O; we call them *observational equivalence classes*.

O_1 is *more powerful* than O_2 if it can distinguish more:
Observational expressiveness: O_1 is *more powerful* than O_2 if the observational equivalence classes for O_1 are strictly finer than those for O_2. It is easy to see that the finest possible equivalence relation over a class C of devices with behavior B is induced by B itself. ■

This notion of expressiveness allows us to show that PTM computations are more expressive than sequences of TM computations, by establishing an *Observational Expressiveness Hierarchy* for PTMs (section 6.3).

6.3 Observational Expressiveness Hierarchy

In this section, we show that PTM computations are more expressive than sequences of TM computations, by establishing an *Observational Expressiveness Hierarchy* for PTMs.

For any k, we define a finite PTM observer Ok as follows:
Finite PTM observer O_k: for any k and any PTM M, O_k(M) is the set of M's observations of length $\leq k$, where the length of an observation is the number of pairs in it. ■

O_k gives us a relative notion of M's behavior, with respect to observers that can only "remember" up to *k* interactions at a time. It is easy to see that for any *k*, O_k is observationally correct:
Proposition 6.3.1: for any pair of PTMs M_1 and M_2, if $B(M_1) = B(M_2)$ then $O_k(M_1) = O_k(M_2)$ ■

It is also easy to see that for any k, O_{k+1} is at least as powerful as O_k:
Proposition 6.3.2: for any k, O_{k+1} is at least as powerful as O_k. ■

It is furthermore the case that for any k, O_{k+1} is more powerful than O_k. In order to prove it, we will construct a family of PTMs $M = (M_1, M_2, ...)$ such that for any k, O_k cannot distinguish between M_k and M_{k+1} whereas O_{k+1} can:
Lemma 6.3: There exists a family of PTMs $M = (M_1, M_2, ...)$ such that for any k, O_k cannot distinguish between M_k and M_{k+1} whereas O_{k+1} can.
Proof: The proof is by construction. All M_i are binary, with inputs 0 or 1, and outputs T and F. At every computational step, for any k, M_k outputs T if the last k input tokens were 1's, F otherwise (if it has not seen that many values, or if some of them were 0). For any k, the shortest distinguishability certificate has length k+1: ((0, F), $(1, F)^{k-1}$, (1, T)). This certificate is consistent with M_k, but not with M_{k+1} ■

Corollary: For any k, O_{k+1} is more powerful than O_k. ■

Thus, we have obtained an infinite family Φ of increasingly powerful PTM observers:
$\Phi = (O_1, O_2, ...)$.
Each observer partitions the class of PTMs into observational equivalence classes, which get progressively finer without reaching a limit.

Given PTMs M_1 and M_2, let k be the smallest number such that O_k can distinguish between M_1 and M_2. Then, we say that k is the *observational distance* between M_1 and M_2:
Observational distance: Given PTMs M_1 and M_2, the *observational distance* between M_1 and M_2 is the smallest number k such that O_k can distinguish between M_1 and M_2. ■

Lemma 6.3 shows that there exist pairs of PTMs that are arbitrarily far apart, in terms of observational distance. However, there are subclasses of PTMs where this is not the case. In particular, the observational distance between any two amnesic PTMs is 1 (Proposition 6.1). In addition, amnesic PTM behaviors are equivalent to TM behaviors (section 4). As a result, we obtain the following theorem:
Theorem 6.3: There exists an infinite observational hierarchy for PTM behaviors, with TM behaviors at the bottom of this hierarchy.

6.4. Compactness conjecture for PTM behaviors

Any observer of a class of machines with behavior B that is capable of observing B (section 7) can be considered the *ultimate observer*. For PTMs, the ultimate observer is one as powerful as $\mathcal{L}$, the language of PTMs (section 5):
Ultimate PTM observer (O_{PTM}): for any M_1, M_2 in C, $\mathcal{L}(M_1) = \mathcal{L}(M_2)$ iff $O_{PTM}(M1) = O_{PTM}(M2)$

In section 7, we have constructed a sequence Φ of finite observers that represent increasingly accurate approximations to O_{PTM}. This raises the following conjecture:
Compactness conjecture for PTMs: two PTMs M_1 and M_2 are distinguishable by O_{PTM} iff there exists a finite observer in Φ which distinguishes them.

As a corollary of the Compactness Conjecture for PTMs, infinite observers are not necessary for distinguishing between all PTMs:
Corollary: If O is a finite PTM observer that observes PTM observations of any length, then O is ultimate.

The *Compactness Conjecture for PTMs* is analogous to the *Compactness Theorem for Logic*, which states that an infinite set of formulae is unsatisfiable (has no model) iff there exists a finite set which is unsatisfi-

able (has no model) [End]. We conjecture that these two theorems are in fact related, and that a more general compactness theorem can be formulated in terms of finite model theory, where these theorems are different versions of the general one.

7. Modeling PTM behavior by recursively defined functions.

We have mentioned in section 3.2 that PTM behaviors can also be modeled by recursive functions from input to output streams. In this section, we define such a function ϕ_M for any PTM M. A similar approach can be found in [Bro].

First, we define two functions over pairs, *1st* and *2nd*, returning the 1st and 2nd components of pairs, respectively. We also define three functions over streams, *head*, *tail,* and *append*: *head*(s) returns the first token of a stream s, *tail*(s) returns the rest of the stream (which is also a stream), and *append*(t, s) appends a token t to the beginning of a stream s (returning a new stream).

Let M be a PTM where f_M is the corresponding computable function (section 2.2), f_M: IxW -> Ox W. We define a function $frec_M$ which takes an input stream ι and a state w and returns an output stream; $frec_M$ is defined recursively as follows:

$frec_M(\iota, w) = append(o, frec(tail(\iota), w'))$,

where o and w' are the 1st and 2nd element respectively of $f_M(head(\iota), w)$

$o = 1st(f_M(head(\iota), w))$; $w' = 2nd(f_M(head(\iota), w))$.

Streams are assumed to be infinite: successive invocations of *tail* continue returning new streams. Streams are also assumed be dynamically bound: *head* and *tail* have lazy evaluation semantics. As a result, $frec_M$ cannot be expressed by any Turing-computable function. Its definition is based on coinductive principles [BM], which are more expressive than the inductive principles underlying Turing-computable functions [JR, WG].

Though *frec* returns the output stream properly, it represents the state explicitly among its inputs, so it is not quite what we need. The function we need, ϕ_M, is obtained by currying the initial state w0 into *frec*:

for any x, $\phi_M(x) = frec_M(x, w0)$.

Though functions from streams to streams are not Turing-computable, they provide a functional way of defining PTM equivalence:

Theorem 7: Given two PTMs *M1* and *M2*, $\mathcal{L}(M1) = \mathcal{L}(M2)$ iff $\phi_{M1} = \phi_{M2}$.

Proof: The proof is omitted from this abstract; it makes use of coinductive reasoning and coalgebraic methods [BM, JR]. ■

8. Conclusion

Though PTM behavior cannot be expressed by recursively enumerable sets, computable functions, or TMs, notions of behavior, equivalence, and expressiveness for PTMs can be expressed as extensions of the corresponding notions for TMs. We have developed the methods, formalisms, and tools for studying models of computation for PTMs that can serve as a basis for a more comprehensive theory of interactive computation.

9. Dedication and acknowledgments

This work is dedicated to Paris Kanellakis, Dina Goldin's late advisor and Peter Wegner's colleague whose spirit guided this work. We would also like to thank Franco Preparata for discussions related to the automata-theoretic aspects of this work.

10. Bibliography

[Abr] Samson Abramsky, Semantics of Interaction, in *Semantics and Logic of Computation*, ed A. Pitts and P. Dibyer, Cambridge, 1997.

[AB] Malcolm Atkinson, Peter Buneman, Types and Persistence in Database Programming Languages, *ACM Computing Surveys* 19:2, 1987.

[BM] Jon Barwise and Lawrence Moss, *Vicious Circles*, CSLI Lecture Notes #60, Cambridge University Press, 1996.

[Bro] Manfred Broy, Compositional Refinement of Interactive Systems, *Digital Systems Research Center SRC 89*, 1992

[End] H. Enderton, *A Mathematical Introduction to Logic*, New York: Academic Press, 1972.

[GW] Dina Goldin, Peter Wegner, *Persistent Turing Machines*, Brown University Technical Report, 1998.

[HU] John Hopcroft, Jeffrey Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley 1979.

[JR] Bart Jacobs and Jan Rutten, A Tutorial on Coalgebras and Coinduction, EATCS Bulletin 62, 1997.

[Kos] Sven Kosub, *Persistent Computations,* Theoretische Informatik Tech. Report, U. Wurzburg, 1998.

[Mil] Robin Milner, Operational and Algebraic Semantics of Concurrent Processes, *Handbook of Theoretical Computer Science*, J. van Leeuwen, editor, Elsevier, 1990.

[MP] Z.Manna, A.Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag 1992

[RN] Stuart Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach,* Addison-Wesley, 1994.

[RS] Michael Rabin, Dana Scott, Finite Automata and their Decision Problems, *IBM J. Res*. 3:2, 1959.

[TB] B.A.Trakhtenbrot, Y.M. Bardzin, *Finite Automata: Behavior and Synthesis*, American Elsevier, 1973.

[Tu] Alan Turing, On Computable Numbers with an Application to the Entscheidungsproblem, *Proc. London Math Society*, 2:42, 1936.

[We1] Peter Wegner, Why Interaction is More Powerful Than Algorithms, *CACM*, May 1997.

[We2] Peter Wegner, Interactive Foundations of Computing, *Theoretical Computer Science*, Feb. 1998.

[WG] Peter Wegner, Dina Goldin, Coinductive Models of Finite Computing Agents, Proc. Coalgebra Workshop (CMCS '99), *Electronic Notes in Theoretical Computer Science*, Vol. 19, March 1999.