V. Interactive Computing

Traditionally, a computational task is considered successful only if it halts after some finite amount of time. However, if we insist on this simple requirement, most modern computer-based equipment will be considered as a failure. For example, a computer-controlled mechanical respiration support system cannot stop operating, or else the patient being supported by this system will die. In addition, operating systems and word processors are written to receive unbounded input over time and therefore do not halt. Also, a rover maneuvering on the surface of another planet should not stop operating before its projected life expectancy,[1] or else it will be considered a (partial) failure.

Apart from this, another aspect of modern computing is that many programs do not compute any function at all. For instance, one may wonder what function is computed by a web server or an FTP client? Quite naturally, one may respond to this question that any web server actually computes some bizarre, huge, unwieldy function. But by going one step further, one can assume that walls, chairs, and even fish tanks ompute such functions (and I will discuss these ideas later). However, one should note that a web server may crash because of a power failure, a random attack, or just because the system administrator shuts it down for maintenance. Since such events are usually not scheduled, one cannot possibly conclude that web severs actually compute something, unless one is a mystic. These and similar observations make it clear that the classical foundations of computer science are inadequate, because they fail to capture many characteristics of modern computer systems. One way out of this dilemma is to introduce interaction into our formal apparatus, and that is exactly the subject of this chapter–how interaction broadens the concept of computability.

5.1 Interactive Computing and Turing Machines

Let us start with a simple question: is the Turing model of computation sufficient to explain and describe modern computer systems? The answer

1. For a system with practically unlimited power supply, this is not really an issue.

is clearly no. The inadequacy of the classical model was briefly presented in the introductory chapter and the paragraph above; however, it is necessary to provide rigorous arguments in favor of the deficiency of this model in describing and explaining the functionality of modern computer systems. Clearly, the adoption of the Turing machine as a complete model for algorithms and general problem-solving lies at the heart of the problem. Most readers will agree with Lynn Andrea Stein's [192] remark that

> Computation is a function from its inputs to its outputs. It is made up of a sequence of functional steps that produce–at its end–some result that is the goal. This is what I was taught when I was trained as a computer scientist. It is a model that computer scientists by and large take for granted. It is something the members of the field share.

To be fair, Stein is not a proponent of this point of view. On the contrary, she is in line with Peter Wegner and Eugene Eberbach, who claim [217] that

> The T[uring]M[machine] model is too weak to describe properly the Internet, evolution or robotics, because it is a closed model... In the case of the Internet, the Web clients "jump" into the middle of interaction, without a knowledge of the server state and previous history of other clients. A dynamic set of inputs and outputs, parallel interaction of multiple clients and servers, a dynamic structure of the Internet communication links and nodes, is outside what a sequential, static and requiring full specification Turing machine can represent.

Of course, Wegner has expressed similar thoughts in other instances (e.g., see [218, 215]), but this is the most recent account of these ideas. Similar views are expressed in a milder tone by Jan van Leeuwen and Jiři Wiedermann [208]:

> The given examples of interactive and global computing indicate that the classical Turing machine paradigm should be revised (extended) in order to capture the forms of computation that one observes in the systems and networks in modern information technology.

In addition, the classical model artificially imposes limits to what can be done with modern computers. Here is what Stein has to say about this [192]:

> Increasingly, however, the traditional computational metaphor limits, rather than empowers, us today. It prevents us from confronting and working effectively with computation as it actually occurs. This is true both within computer science, which still clings fervently to the metaphor, and in other disciplines where

> dissatisfaction with the computational metaphor [i.e., the idea that the brain is like a computer] has in some cases caused an anticomputationalist backlash.

It is really debatable whether the anticomputationalist backlash was caused by a dissatisfaction with the classical computational metaphor. On the contrary, computationalism asserts that *all* mental processes are mechanical in their nature. In other words, according to computationalism, mental processes can be implemented by either computers or hypercomputers. And precisely this is the reason for the anticomputationalist backlash. Apart from this, the essence of the whole argument is that the traditional model of computation is clearly inadequate for modern computer practice. For example, young students who are trained to program sequentially on systems with a single processor, are not adequately prepared for the real world, in which most modern programming tasks involve some kind of concurrency, and in many cases one has to implement "algorithms" on machines with more than one processor.

A typical counterargument to the previous rhetoric would be that all parts of various interactive systems can be modeled by Turing machines. However, a Turing machine always computes a result; but then, one should be able to answer the following question raised by Stein: "What is it that the world-wide web calculates?" In addition, one may ask what an Internet relay chat server computes. Naturally, one may raise similar questions for many other instances in which modern computer equipment is in use today. Our inability to give convincing answers to such questions is a clear indication that the Turing machine is an outdated model of computation that has a very limited role to play in modern-day computing.

Bertil Ekdahl [56] argues that interactive computing can be simulated by oracle Turing machines. However, this argument is clearly a fallacy: The oracle of a Turing machine contains quite specific information (e.g., the characteristic function of a set) that is used in the course of the operation of the machine. Thus, one may say that an oracle machine is thereby able to communicate with the external world. But a typical interactive system has bidirectional communication with the environment, which is not the case for an oracle machine. Also, oracle machines have all the drawbacks of ordinary Turing machines that make then inadequate as models of interactive systems (i.e., they expect their input at the beginning of the computation and succeed in computing something only when they stop). In addition, one should not forget that interaction is a primitive notion (e.g., the π-calculus, which was introduced by Robin Milner, Joachim Parrow, and David Walker [133], was built around this primitive notion), just as the notion of sequentially reading from and writing to a storage medium is a primitive notion in which the Turing machine model rest.

Doug Lea [109] remarks that just as "the few constructs supporting sequential programming lead to a wide range of coding practices, idioms, and

design strategies, a few concurrency constructs go a long way in opening up new vistas in programming." Consequently, as Stein notes, if we discourage (or even prevent) students from adopting certain styles of thinking and understanding just because they deviate from the unrelenting sequentialism of the computational metaphor, students are not learning new coding practices, idioms, and design strategies, and eventually become ill prepared for today's software market.

If Turing machines are inadequate for describing modern computer systems, then we clearly need other formalisms that can deal with the various aspects of these systems. Indeed, there have been a number of calculi and/or conceptual devices that address these issues. In the rest of this chapter I am going to give an overview of some of them.

5.2 Interaction Machines

When Turing proposed his famous machine, he actually set the foundations of sequential computing. However, in the case of interactive computing, things proceeded in the opposite direction. First, programmers implemented interactive systems and practiced interactive programming, and only then did theoreticians start to formulate theories that dealt with certain aspects of interactive computing (for example, see [4, 129]). However, an integrated theory of interactive computing appeared only in 1998 when Wegner published his paper "Interactive foundations of computing" [216].

In this paper Wegner discusses the basic characteristics of *interaction machines* as well as their computational power. In addition, he presents "interaction grammars" that extend the Chomsky hierarchy of grammars.

An interaction machine is simply a Turing machine that is augmented with the capability of performing dynamic read and/or write actions that provide it with a way of directly interacting with the external world. This additional capability can be implemented by allowing interaction machines to be connected with their environment–more specifically, with a single or multiple input stream or via synchronous or asynchronous communication. From a practical point of view this means that there is no single definition of the structure of an interaction machine. Moreover, all interaction machines are open systems. The *observable* behavior of interaction machines is specified by *interaction histories*, which take the form of *streams* that are the interactive time-sensitive analogue of strings. Formally, if A is a set, then by a stream over A we mean an ordered pair $s = (a, s')$ where $a \in A$ and s' is another stream.[2] The following statement is a clear indication that the additional "hardware" is not just some kind of accessory.

2. Streams are objects that do not belong to the standard set-theoretic universe, but they do belong to the universe of non-well-founded sets (see [8] for details).

Proposition 5.2.1 *It is not possible to model interaction machines by Turing machines.* [216]

Interaction machines can be viewed as mappings over streams that take time into account. Such mappings cannot be classified as functions, since functions are timeless. In other words, interaction machines extend the theory of computability by introducing *computable nonfunctions* over histories. This nonfunctional facet of interaction machines applies also to other aspects of these machines. This can be demonstrated by a simple example: consider a rover maneuvering on the surface of another planet. Clearly, the rover is an interactive system that must respond to external stimuli. For instance, when it encounters a boulder it must change its course; when it is going down into a crater it must use its brakes to reduce its speed, while when it is going up a hillside it must boost its engines. In many cases, the software loaded into the rover's memory cannot handle totally unexpected situations, and so a new, updated software is uploaded to the rover. None of these actions can be predicted, and sometimes they are not among the actions one initially expects the rover to face. In other words, these actions cannot be described by a function, and thus one may say that they cannot be described algorithmically. From this example it is not difficult to see that the behavior of interaction machines cannot be described by Turing computable functions. An interesting and, to some degree, unexpected effect of interaction is that if we enhance algorithms with interactive behavior, we create systems that operate in a smart (not intelligent!) way, or in Wegner's own words, "[e]xtending algorithms with interaction transforms dumb algorithms into smart agents." The crux of the ideas presented so far have been summarized by Wegner in the form of a thesis.

Thesis 5.2.1 ***Inductive computing:*** *Algorithms (Turing machines) do not capture the intuitive notion of computing, since they cannot express interactive computing and intuitive computing includes interaction* [216].

Before we proceed with the presentation of interaction grammars, we will briefly recall the definitions of formal grammar and the Chomsky hierarchy of grammars, as well as their relationship to various forms of automata. Readers familiar with these notions can safely skip this material. Assume that Σ is an arbitrary set and that ε denotes the empty string. Then

$$\Sigma^* = \{\varepsilon\} \cup \Sigma \cup \Sigma \times \Sigma \cup \Sigma \times \Sigma \times \Sigma \cup \cdots .$$

is the set of all finite strings over Σ. Let us now recall the definition of a grammar.

Definition 5.2.1 A grammar is defined to be a quadruple $G = (V_N, V_T, S, \Phi)$ where V_T and V_N are disjoint sets of terminal and nonterminal (syntactic class) symbols, respectively; S, a distinguished element of V_N, is called the

starting symbol. Φ is a finite nonempty relation from $(V_T \cup V_N)^* V_N (V_T \cup V_N)^*$ to $(V_T \cup V_N)^*$. In general, an element (α, β) is written as $\alpha \rightarrow \beta$ and is called a production or rewriting rule [204].

Grammars are classified as follows:

Unrestricted grammars There are no restrictions on the form of the production rules.

Context-sensitive grammars The relation Φ contains only productions of the form $\alpha \rightarrow \beta$, where $|\alpha| \leq |\beta|$, and in general, $|\gamma|$ is the length of the string γ.

Context-free grammars The relation Φ contains only productions of the form $\alpha \rightarrow \beta$, where $|\alpha| = 1$ and $\alpha \in V_N$.

Regular grammars The relation Φ contains only productions of the form $\alpha \rightarrow \beta$, where $|\alpha| \leq |\beta|$, $\alpha \in V_N$, and β has the form aB or a, where $a \in V_T$ and $B \in V_N$.

Syntactically complex languages can be defined by means of grammars. To each class of languages there is a class of automata (machines) that *accept* (i.e., they can answer the decision problem "$s \in L$?," where s is a string and L is a language) this class of languages, which are generated by the respective grammars. In particular, *finite automata* accept languages generated by regular grammars, *push-down automata* accept languages generated by context-free grammars, *linear bounded automata* accept languages generated by context-sensitive grammars, and *Turing machines* accept *recursive* languages, that is, a subclass of the class of languages generated by unrestricted grammars.

An interaction grammar is not used to recognize strings but rather streams defined above.

Definition 5.2.2 An interaction grammar IG is defined to be a quadruple (V_N, V_T, S, R), where V_N, V_T, and S have their "usual" meaning and R is a set of production rules. Given a production rule $\alpha \rightarrow \beta$, β may be formed using the "listening" operator . and the "nondeterministic choice" operator +.[3]

Generally speaking, the . operator waits for input, while the + operator selects nondeterministically an event from a list of events when input arrives. Thus, an interactive grammar containing only the production rule

$$\text{BinDigit} \rightarrow (0 + 1).\text{BinDigit}$$

describes infinite streams, expressing reactive systems that react to a continuous (nonhalting) stream of zeros and ones over time.

3. Although it is not explicitly stated, one may use parentheses for clarity.

It is known that one can compose sequential processes and create a new process that has the combined effect of the two processes. Practically, this means that one can compose two Turing machines to create paired Turing machines that compute exactly what the two distinct machines compute. On the other hand, it is not possible to compose interaction machines in a similar way. However, we can combine interaction machines by means of the parallel composition operator, denoted by $|$. Thus, the behavior of $P|Q$ is "equal" to the behavior of P, the behavior of Q, and the interaction that takes place between P and Q. It is interesting to note that parallel composition is a commutative operation, that is, $A|B = B|A$.

Interactive identity machines are a special form of interaction machines that immediately output their input without transforming it. These machines can express richer behavior than Turing machines, because they trivially model Turing machines by simply echoing their behavior. Interactive identity machines can be used to model "echo intelligence" (a behavior that is best exemplified by the legendary Eliza program by Joseph Weizenbaum [221]).

5.3 Persistent Turing Machines

Persistent Turing machines, which were introduced by Dina Goldin [69], are extended Turing machines that can describe a limited form of interactive behavior. In particular, they can be employed to describe *sequential interactive computations* that are applied to a dynamic stream consisting of input/output pairs and have their state stored in some medium [70]. A persistent Turing machine is a Turing machine that operates on a number of different tapes. In addition, the contents of a distinguished tape, which is called the *persistent work tape* (or just work tape), are preserved between any two complete computational tasks. This distinguished tape plays the role of the permanent *memory* of the machine, and its contents specify the *state* of the machine before and after a computation. The states of a persistent Turing machine are represented by strings with no restriction on their length (i.e., they may even be infinite).

A persistent Turing machine $\mathscr{P}$ defines a partial recursive function $\varphi_{\mathscr{P}}: I \times W \to O \times W$, where I, O, and W denote its input, output, and work tape. To demonstrate how we can define this function, I will borrow an example from [69]. A telephone answering machine $\mathscr{A}$ is actually a persistent Turing machine that defines the following function:

$$\begin{aligned} \varphi_{\mathscr{A}}(\text{record } x, y) &= (\text{ok}, yx), \\ \varphi_{\mathscr{A}}(\text{play back}, x) &= (x, x), \\ \varphi_{\mathscr{A}}(\text{erase}, y) &= (\text{done}, \varepsilon). \end{aligned}$$

The answering machine can record, play back, and erase messages. In addition, the work tape of the answering machine contains a stream of recorded messages. Thus, $\varphi_{\mathscr{A}}$ fully describes the observable behavior of the answering machine. Notice that the contents of the work tape are only part of the definition of $\varphi_{\mathscr{A}}$ and by no means affect the behavior of $\mathscr{A}$.

Let us summarize: a persistent Turing machine $\mathscr{P}$ transforms an input stream $(i_1, i_2, \ldots)$ to an output stream $(o_1, o_2, \ldots)$ using a function $\varphi_{\mathscr{P}}$. Initially, the state of $\mathscr{P}$ is empty. In the course of its operation the state of $\mathscr{P}$ changes. For instance, in the case of our answering machine, a possible input stream may be

$$(\text{record } A, \text{record } BC, \text{erase}, \text{record } D, \text{record } E, \text{play back}, \ldots).$$

This input stream generates the output stream

$$(\text{ok}, \text{ok}, \text{done}, \text{ok}, \text{ok}, DE, \ldots),$$

while the state evolves as follows:

$$(\varepsilon, A, ABC, \varepsilon, D, DE, DE, \ldots).$$

Assume that I and O are the input and output streams of a persistent Turing machine $\mathscr{P}$. Then the *interaction stream* of $\mathscr{P}$ is a stream of pairs, where the first part of each pair comes from the input stream and the second part from the output stream. Thus, the interaction stream of our answering machine has the following form:

$$\Big((\text{record } A, \text{ok}), (\text{record } BC, \text{ok}), (\text{erase}, \text{done}), (\text{record } D, \text{ok}), \ldots\Big).$$

In order to compare two different conceptual computing devices, Goldin uses the notion of behavioral equivalence. Two conceptual computing devices are equivalent if they have the same behavior. In the case of string-manipulating devices (e.g., ordinary Turing machines), the collection of strings that are processed and generated by the device constitutes its behavior. For example, the behavior of a Turing machine is formed by the strings that are read and printed by its scanning head. More generally, the behavior of a conceptual computing device $\mathscr{D}$ can be modeled by its corresponding language $\mathcal{L}(\mathscr{D})$. For instance, for any persistent Turing machine $\mathscr{P}$, the set of all interaction streams makes up its language $\mathcal{L}(\mathscr{P})$. Since the language of a conceptual computing device models its behavior, one can say that two such devices $\mathscr{D}_1$ and $\mathscr{D}_2$ are equivalent if $\mathcal{L}(\mathscr{D}_1) = \mathcal{L}(\mathscr{D}_2)$.

A persistent Turing machine processes an arbitrary input stream that is generated by its environment. Clearly, this is not a realistic assumption, since the external environment cannot yield an arbitrary input stream (e.g., the winning numbers in a lottery drawing are usually in the range 1 to 48, and these numbers are the input for the lottery players). This remark has led Goldin to a general definition of equivalence in the following way.

Definition 5.3.1 Assume that C is a set of conceptual computing devices and B a function that returns the behavior of some machine. Then an environment O for C is a function $O : C \to \beta_O$ that is *consistent*, which means that

$$\forall \mathscr{M}_1, \mathscr{M}_2 \in C : B(\mathscr{M}_1) = B(\mathscr{M}_2) \Rightarrow O(\mathscr{M}_1) = O(\mathscr{M}_2).$$

The elements of β_O are called *feasible behaviors* (within the environment O). If $O(\mathscr{M}_1) \neq O(\mathscr{M}_2)$, then $\mathscr{M}_1$ and $\mathscr{M}_2$ are *distinguishable* in O; otherwise, they *appear equivalent* in O.

Given a set C of conceptual computing devices with behavior B, then any environment O for C can be used to partition C into equivalence classes. Each of these classes is called a *behavioral equivalence class*, since the members of each equivalence class appear equivalent in O. Based on this, it is possible to classify environments.

Definition 5.3.2 Given two environments O_1 and O_2, then O_1 is *richer* than O_2 if its behavioral equivalence classes are strictly finer that those of O_2.

Quite naturally, it is possible to define an infinite sequence Θ of finite persistent Turing machines' environments $\Theta = (O_1, O_2, \ldots)$, provided that for any k, $O_k(\mathscr{M})$ is the set of prefixes of interaction streams, of length less than or equal to k. It can be proved that for any such sequence, O_{k+1} is richer than O_k. The main result concerning environments is the following.

Theorem 5.3.1 *The environments in Θ induce an infinite expressiveness hierarchy of persistent Turing machine behaviors, with Turing machines at the bottom of the hierarchy* [69].

Goldin admits that the behavior of any persistent Turing machine is not rich enough to describe an arbitrary interactive system. However, as mentioned above, these machines can be used to describe any sequential interactive computation. This observation has been formulated in [70] as follows.

Thesis 5.3.1 *Any sequential interactive computation can be performed by a persistent Turing machine.*

On the other hand, the behavior of a Turing machine is at the bottom of the expressiveness hierarchy, which simply implies that Turing machines are an inadequate model of computation for modern computer equipment.

5.4 Site and Internet Machines

Site and Internet machines were introduced by van Leeuwen and Wiedermann [209] to model individual machines, possibly connected with other

machines, and a network of site machines. More specifically, a site machine models a normal personal computer that is equipped with a hard disk having potentially unlimited capacity. A site machine can communicate with its environment by sending and receiving messages via a number of ports. One may think of a site machine as a random-access machine that can use sockets to communicate with its environment. The messages that a site machine may receive or send consist of symbols from a finite alphabet Σ. The special symbol $\tau \in \Sigma$ is used to designate the end of some communication. One may think that τ is something like the ASCII EOT (End Of Transmission) character that signals the end of the current transmission.

Typically, the hardware and/or software of a site machine can be changed by an external operator called an *agent*. The agent is part of the environment and communicates with a site machine via its ports. As in real life, when the configuration of the machine changes (e.g., when it is being maintained by the agent), either the machine is temporarily switched off or its communication with the environment is temporarily blocked. Since site machines are equipped with a permanent memory, no data is lost during hardware or software upgrades. When the upgrade process is finished, the machine will be able to resume its operation and consequently, its communication with the environment. It is quite possible to define a function γ that returns a description of the hardware or software upgrade that is taking place at time t. If no such operation is taking place, one may assume that γ returns an empty string. Generally speaking, the function γ is *noncomputable* (in the classical sense of the word) and its values are not a priori known. These remarks are justified because one cannot foresee the actions of any agent. In other words, one cannot tell beforehand what might go wrong with a computer system. If we could actually compute such a function, then the notion of a computable future would be no exaggeration!

A site machine performs a computation by transforming an infinite multiplex input stream into a similar output stream. More specifically, if a site machine has n input ports and m output ports, it processes a stream of n-tuples to produce a stream of m-tuples. In other words, a site machine computes mappings Φ of the form $(\Sigma^n)^\infty \rightarrow (\Sigma^m)^\infty$. Note that if A is an alphabet, then A^ω denotes the set of infinite strings over the alphabet A. Also, $A^\infty = A^* \cup A^\omega$, which is the set of finite and infinite strings over the alphabet A.

Clearly, a site machine is not a basic conceptual computing device. Thus, if one wants to study the computational power of site machines, it is necessary to design a conceptual computing device that mimics the behavior of a site machine. Most attempts to define new conceptual computing devices are based on the Turing machine. In general, this approach is based on the conservative idea that the Turing model is simple and valid, so all extensions should be based on it. Thus, we are going to construct a new conceptual device that is basically a Turing machine augmented with a number of new features. More specifically, our new extended Turing machine will

be equipped with three new features: *advice*, *interaction*, and *infinity of operation*.

Any candidate mechanism that models the change of software or hardware must satisfy the following two requirements:

(i) changes should be independent of the current input read by the machine up to the moment of the actual change, and

(ii) changes should not be large.

These requirements can be met once we demand that the description of new hardware or software depend only on the moment t it actually happens. In addition, the size of the description has to be "at most polynomial in t" (i.e., it has to be reasonably short).

Oracles can be used to enter new, external information into the machine. However, "ordinary" oracles are too "loose" for our case, and so van Leeuwen and Wiedermann have opted to use *advice functions*. Turing machines with advice were studied by Karp and Lipton [94]. Assume that $S \subset B$, where $B = \{0,1\}^*$. In addition, suppose that $h : \mathbb{N} \to B$. Next, we define the set

$$S : h = \left\{ wx \;\middle|\; \Big(x \in S\Big) \wedge \Big(w = h(|x|)\Big) \right\}.$$

Recall that $|x|$ denotes the length of the bit string x. Let $\mathcal{S}$ be any collection of subsets of B. Also, let $\mathcal{F}$ be any collection of functions from the set of natural numbers to the set of natural numbers. Then

$$\mathcal{S}/\mathcal{F} = \left\{ S \;\middle|\; (\exists h)\Big((\lambda n.|h(n)| \in \mathcal{F}) \wedge (S : h \in \mathcal{S})\Big) \right\}.$$

The intuitive meaning of $\mathcal{S}/\mathcal{F}$ is that it is the collection of subsets of B that can be accepted by $\mathcal{S}$ with $\mathcal{F}$ "advice." In this book we will be concerned only with the class P/poly. Notice that the P/poly class of languages is characterized by a Turing machine that receives advice whose length is polynomially bounded and computes in deterministic polynomial time. By substituting the set $\{0,1\}$ with Σ, one may get similar definitions and results.

In order to make a Turing machine with advice able to interact with its environment, we must equip it with a (finite) number of input and output ports. In addition, in order to accommodate infinite computation, one may consider the modus operandi of infinite-time Turing machines. Having roughly specified how advice, interaction, and infinity of operation can be accommodated in a single conceptual computing device, we need to give a description of how the resulting machine will operate. Initially, the tapes of the machine are assumed to be filled with blanks. In addition, the machine's operation depends on some controlling device. At each step, the machine reads the symbols that appear in its input ports and writes some symbols

to its output ports. What the machine will do next depends on what it has read, what lies under its scanning heads, and the instruction being executed. Also, at any time t the machine can consult its advice only for values of $t_1 \leq t$. Machines that have these characteristics are termed *interactive Turing machines with advice.*

Theorem 5.4.1 *For every site machine there exists an interactive Turing machine with advice $\mathscr{A}$ that has the same computational power. In addition, for every interactive Turing machine with advice there exists a site machine that has the same computational power.*

The following theorem makes precise the equivalence stated in the previous theorem.

Theorem 5.4.2 *Assume that $\Phi : (\Sigma^n)^\infty \rightarrow (\Sigma^m)^\infty$, $n, m > 0$, is a function. Then the following statements are equivalent:*

(i) The function Φ can be computed by a site machine.

(ii) The function Φ can be computed by an interactive Turing machine with advice.

The Internet is the international computer network of networks that connects government, academic, and business institutions. Every machine that is part of the Internet has its own address. Internet machines are a model of the Internet. As such, an Internet machine consists of a number of different site machines. All machines that make up an Internet machine have their own unique addresses. As in the case of a simple network, we need to know which machines are active at any given moment. Thus, we define a function α that returns the list of addresses of those machines that are active at time t. In addition, we can safely assume that for all t, the size of the list $\alpha(t)$ is polynomially bounded.[4] Also, we assume that the site machines making up an Internet machine operate asynchronously and communicate by exchanging messages.

Typically, an IP packet is a chunk of data transferred over the Internet using the standard Internet protocol. Each packet begins with a header containing the address of the sender, the address of the receiver, and general system control information. Similarly, the header of any message that site machines exchange contains the address of both the sender and the receiver. Naturally, it is unnecessary to include any system control information, since we are defining a conceptual device in the broad sense of the term. In the real world, it is impossible to predict the amount of time it takes for a message to arrive at a destination machine from the moment it

4. A function $f(n)$ is polynomially bounded if $f(n) = O(n^k)$ for some constant k. Practically, this means that there are positive constants c and l such that $f(n) \leq cn^k$ for all $n \geq l$. The values of c and l must be fixed for the function f and must not depend on n.

has been sent, not to mention the possibility that the message never gets delivered. This implies that the time it takes for a message emitted by a site machine to reach its destination should not be predictable. However, one can give an estimate of this time. Thus, at any given moment t, for any two site machines $i,j \in \alpha(t)$, one can define a function β that will "compute" the estimated delivery time. For messages that are addressed to some machine $k \notin \alpha(t)$, the sending machine will receive an error message just after the message has been sent. Clearly, not all machines are directly connected and thus the message is actually sent from a machine that resides in the proximity of where the non-existing machine is supposed to be. Messages that have the same (existing) recipient enter a queue if they arrive at same time, and consequently, they will be processed accordingly. At any time t and for any machine $i \in \alpha(t)$, the function γ returns a (formal) description of the hardware or software upgrade that might take place at t on machine i.

Functions α, β, and γ fully specify the operation of a given Internet machine. Generally speaking, these functions are noncomputable and their return values are "computed" by consulting a number of finite tables.

It is not hard to see that Internet machines compute mappings that are similar to those that can be computed by site machines. However, since an Internet machine consists of a number of site machines that may have different numbers of input and output ports, this obviously affects the mappings that can be computed by a given Internet machine. Without getting into the technical details, one can prove that for every Internet machine there exists an interactive Turing machine with advice that *sequentially* realizes the same computation as the Internet machine. Clearly, the opposite also holds true.

Site and Internet machines are conceptual computing devices that are supposed to model our personal computers and the Internet, respectively. Both these conceptual computing devices seem to transcend the capabilities of the Turing machine. They seem to transcend even the capabilities of interaction machines. In spite of the fact that it has not been directly demonstrated how these machines can tackle classically unsolvable problems, we still classify them provisionally as hypermachines, since they seem to transcend the capabilities of Turing machines.

5.5 Other Approaches

If we assume that the Church–Turing thesis is indeed valid, then for every effectively computable function f there is a λ-term and vice versa. Let us now hypothesize that there exists a calculus that is built around a notion more "fundamental" than the corresponding notion on which the λ-calculus is built. Also, assume that this new calculus is general enough so one can simulate the λ-calculus within it, but at the same time, it is impossible

to simulate this new calculus within the λ-calculus. Clearly, this hypothetical new calculus is more expressive than the λ-calculus. In addition, it would be interesting to see whether classically noncomputable functions become "computable" in this new framework, provided that we are able to define an equivalent model of computation. The very existence of such a calculus, and its accompanying model of computation, would affect the validity of the Church–Turing thesis. Naturally, a direct consequence would be a "broadening" of the thesis.[5] The most important question is whether such a calculus actually exists.

The π-calculus [132] is a process algebra that is built around the primitive notion of interaction. The calculus is particularly well suited for the description of systems in which *mobility*[6] plays a central role. In addition, the π-calculus has as a special case the λ-calculus [130]. In other words, for every λ-term there is an "equivalent" π-calculus process expression, but not vice versa. Moreover, if there are two λ-terms that are equated using λ-calculus means, their translations can be distinguished in the π-calculus, which makes the π-calculus strictly more expressive than the λ-calculus [19].

Let us now see how we can translate a λ-term into the π-calculus. Since this translation is purely syntactic, we need to briefly review the syntax of both calculi. The set of π-calculus process expressions is defined by the following abstract syntax:

$$P ::= \Sigma_{i\in \mathrm{I}}\pi_i.P_i \;\Big|\; P_1|P_2 \;\Big|\; \text{new } \alpha\ P \;\Big|\; !P\ .$$

If $\mathrm{I} = \emptyset$, then $\Sigma_{i\in I}\pi_i.P_i = \mathbf{0}$, which is the null process that does nothing. In addition, π_i denotes an *action prefix* that represents either sending or receiving a message, or making a silent transition:

$$\pi ::= x(y) \;\Big|\; \bar{x}\langle y\rangle \;\Big|\; \tau\ .$$

The expression $\Sigma_{i\in \mathrm{I}}\pi_i.P_i$ behaves just like one of the P_i's, depending on what messages are communicated to the composite process; the expression $P_1|P_2$ denotes that both processes are concurrently active; the expression new α P means that the use of the message α is restricted to the process P; and the expression $!P$ means that there are infinitely many concurrently active copies of P.

The set of λ-terms is defined by the following abstract syntax:

$$M ::= x \;\Big|\; \lambda x.M \;\Big|\; MN\ .$$

We are now ready to present the translation of any λ-term into the π-calculus.

5. This is true, as evidenced by the fact that the authors of the various models of interactive computation presented in this chapter have reformulated the Church–Turing thesis.
6. The term "mobility" here means among others things that processes may move in a virtual space of linked processes or that processes move in a physical space of computing sites.

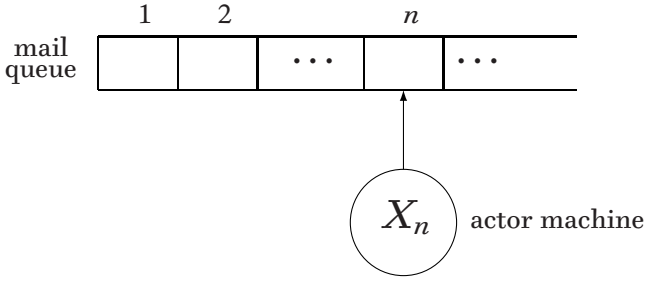

Figure 5.1: An abstract representation of an actor.

Definition 5.5.1 Assume that M is an arbitrary λ-term. Then its translation $[\![M]\!]$ into the π-calculus is an abstraction defined inductively as follows:

$$
\begin{aligned}
[\![x]\!](u) &\stackrel{\text{def}}{=} \bar{x}\langle u\rangle, \\
[\![\lambda x.M]\!](u) &\stackrel{\text{def}}{=} u(xv).[\![M]\!]\langle v\rangle, \\
[\![(MN)]\!](u) &\stackrel{\text{def}}{=} \text{new}\, v\Big([\![M]\!]\langle v\rangle \mid \text{new}\, x(\bar{v}\langle xu\rangle \mid\, !x[\![N]\!])\Big).
\end{aligned}
$$

Notice that in the last equation, x is a bound name in N. Also, $[\![M]\!](u)$ denotes that $[\![M]\!]$ is actually an abstraction that is applied to an argument list u.

The π-calculus is not really a model of computation; it is rather a mathematical theory with which one can describe the functionality of computational models or systems. A theory that is closer to what one may call a true model of interactive computation is the *actors* model of concurrent computation created by Carl Hewitt, Henry Baker, and Gul Agha [4]. The actors model is an untyped theory that is a generalization of the λ-calculus. Actors communicate with each other by sending messages. Incoming communication is mapped to a triple that consists of:

(i) a finite set of messages sent to other actors,

(ii) a new behavior that is a function of the communication accepted (and thus the response to the next communication depends on it), and

(iii) a finite set of newly created actors.

Each actor has its own mail address and its own mail queue, with no limit on the number of messages it can hold. Notice that the behavior of an actor is determined by the relationships among the events that are caused by it. Also, it is rather important to note that an actor is defined by its behavior and not by its physical representation. Figure 5.1 depicts an abstract representation of an actor. The information contained in the actor machine determines the behavior of the actor; thus, it can be sensitive to history.

Actors are a model of computation that is more powerful than the classical model of computation. For instance, it is not difficult to simulate arbitrary sequential processes (e.g., Turing machines) or purely functional systems based on the λ-calculus by a suitable actor system, but it is not possible to simulate an actor system by a system of sequential processes. The reason why the converse is not possible is the ability of any actor to create other actors. And this is one of the reasons the π-calculus is more expressive than the λ-calculus: by using the replication operator, !, one can specify the generation of an unbounded number of copies of a given process.