III. Early Hypercomputers

Hypercomputation is not really a recent development in the theory of computation. On the contrary, there were quite successful early efforts to define primarily conceptual computing devices with computational power that transcends the capabilities of the established model of computation.[1] In this chapter, I will present some of these early conceptual devices as well as some related ideas and theories. In particular, I will present trial-and-error machines, TAE-computability, inductive machines, accelerated Turing machines, oracle machines, and pseudorecursiveness. However, I need to stress that I have deliberately excluded a number of early efforts, which will be covered later in more specialized chapters.

3.1 Trial-and-Error Machines

In this section I present the theory of trial-and-error machines, a model of the human mind based on these machines, and TAE-computability, a model of computation that is similar to trial-and-error machines.

3.1.1 Extending Recursion Theory

In 1965, the prestigious *Journal of Symbolic Logic* published in a single issue two papers [68, 160] by Mark Gold and Hilary Putnam that dealt surprisingly with the same subject–*limiting recursion*. This type of recursion can be realized in the form of *trial-and-error* machines. Typically, a trial-and-error machine is a kind of a Turing machine that can be used to determine whether an element x belongs to a set $X \subset \mathbb{N}$ or, more generally, whether a tuple $(x_1, \ldots, x_n)$ belongs to a relation $R \subset \mathbb{N}^n$. In the course of its operation, the machine continuously prints out a sequence of responses (e.g., a sequence of 1's and 0's) and the last of them is always the correct

1. Strictly speaking, Kalmár [93], Rózsa Péter [154], and Jean Porte [156] were probably the first researchers to challenge the validity of the Church-Turing thesis. Nevertheless, their arguments were not without flaws, as was shown by Elliott Mendelson [126].

answer. Thus, if the machine has most recently printed 1, then we know that the integer (or the tuple) that has been supplied as input must be in the set (or relation) *unless the machine is going to change its mind*; but we have no procedure for telling whether the machine will change its mind again. Suppose now that our trial-and-error machine prints out an infinite number of responses. Then after a certain point, the machine may converge to a particular response, and thus it will continuously print out the same response (1 or 0). Of course, this description is somehow vague, and so we need to define precisely limiting recursion. Let us begin with limiting recursive predicates.[2]

Definition 3.1.1 A function P is a limiting recursive predicate if there is a general recursive function f such that (for every $x_1, x_2, \ldots, x_n$),

$$P(x_1, x_2, \ldots, x_n) \iff \lim_{y \to \infty} f(x_1, x_2, \ldots, x_n, y) = 1,$$

$$\neg P(x_1, x_2, \ldots, x_n) \iff \lim_{y \to \infty} f(x_1, x_2, \ldots, x_n, y) = 0,$$

where

$$\lim_{y \to \infty} f(x_1, x_2, \ldots, x_n, y) = k \stackrel{\text{def}}{=} (\exists y)(\forall z)(z \geq y \rightarrow f(x_1, \ldots, x_n, z) = k).$$

The following theorem is proved in [160].

Theorem 3.1.1 *P is a limiting recursive predicate if $P \in \Delta_2^0$.*

Obviously, this means that one cannot use a Turing machine to check whether a limiting recursive predicate P is true or false. Thus, trial-and-error machines transcend the Church–Turing limit.

Assume now that we restrict a trial-and-error machine so it can change its mind only k times, irrespective of the particular input the machine has. As a direct application of this restriction, k-limiting recursion was introduced.

Definition 3.1.2 P is a k-limiting recursive predicate if there is a general recursive function f such that (for every $x_1, x_2, \ldots, x_n$):

(i) $P(x_1, x_2, \ldots, x_n) \iff \lim_{y \to \infty} f(x_1, x_2, \ldots, x_n, y) = 1$;

(ii) there are *at most* k integers y such that

$$f(x_1, \ldots, x_n, y) \neq f(x_1, \ldots, x_n, y + 1).$$

The following theorem is proved in [160].

2. Putnam calls these predicates *trial-and-error* predicates, but we have opted to use Gold's terminology.

Theorem 3.1.2 *There exists a k such that P is a k-limiting recursive predicate if and only if P belongs to* ${}^*\Sigma^0_1$*, the smallest class containing the recursively enumerable predicates and closed under truth functions.*

Limiting recursive functions can be defined similarly to limiting recursive predicates.

Definition 3.1.3 A partial function $f(x)$ will be called limiting recursive if there is a total recursive function $g(x, n)$ such that

$$f(x) = \lim_{n \to \infty} g(x, n).$$

Similarly, one can define limiting recursive sets and relations (see [68] for details).

It has already been noted that trial-and-error machines transcend the capabilities of ordinary Turing machines; thus they should be able to solve the halting problem. Indeed, Peter Kugel [103] describes an *effective method* (or *hyperalgorithm*) that can solve this problem. Here is his effective method to solve this problem:

> Given a program, Prog, and an input, Inp, output NO (to indicate that Prog(Inp) will not halt). Then run a simulation of Prog(Inp). (Turing [206] showed that such a simulation is always possible.) If the simulation halts, output YES to indicate that Prog(Inp) really does halt.
>
> Clearly the last output that this procedure produces solves the halting problem, if you are willing to accept results arrived at "in the limit." Which proves that limiting computation can do things no ordinary, or recursive, computation can.

3.1.2 A Model of the Human Mind

Another aspect of Kugel's work is a proposed model of intelligence, and consequently a model of the human mind, that is based on limiting recursion. In particular, Kugel is a strong advocate of the idea that the human mind is actually a trial-and-error machine. He has suggested a division of the human mind into four parts or modules [102]. Figure 3.1 depicts Kugel's division.

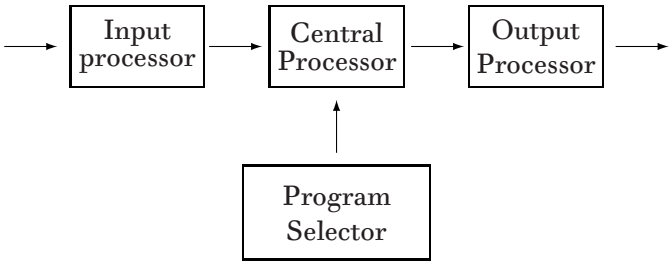

Figure 3.1: Kugel's division of the human mind.

The functionality of each module is briefly outlined below:

Input Processor This module gathers information from the environment and transforms it into a form suitable for further processing by the central processor. For example, suppose that Lila is a zoologist who studies a herd of zebras in an African savanna. Suddenly, she realizes that a tiger is approaching the place where she is standing. At once, her input processor takes this visual signal and turns it into the message, "This is a tiger."

Central Processor The transformed data that the input processor produces are further transformed into a form that is meaningful for the output processor. For example Lila's central processor might transform the message, "This is a tiger" into the message, "run."

Output Processor This module takes the information produced by the central processor and transforms it into something that can be used to affect the world. For example, Lila's output processor might take the message, "run" and turn it into messages to control specific muscles so as to remove Lila from the immediate area.

Program Selector In general, different situations demand different actions. In Kugel's model, the human mind has a set of (predefined?) actions, which he calls *programs*, that can be invoked to handle a particular situation. The program selector is the module that is responsible for the invocation of the appropriate program. For example, Lila's program selector will most probably decide that it is time to invoke the *animal-recognizing program* and halt the *zebra-studying program*.

Kugel argues that the input processor is a Π_1^0 process (i.e., it can be simulated by a machine capable of computing such functions), the output processor is a Σ_2^0 process, the central processor is a k-limiting recursive process, and the program selector is a Π_2^0 process.

Kugel has derived his results by tacitly assuming that all mental processes are mechanistic in nature and part of the arithmetic hierarchy. This

observation explains the nature of his results, but not how he arrived at his conclusions. Therefore, to explain how he did so, I will briefly present the ideas that led him to this particular model of the mind.

Among other things, the input processor classifies what we see or observe. Based on earlier suggestions made by other researchers, Kugel has suggested that our ability to classify what we observe might involve the recognition of membership in productive sets. A set $A \subseteq \mathbb{N}$ is called productive when there is no computing procedure to determine whether some element x belongs to A. In addition, there is a computable function, the production function, that effectively finds a counterexample to the claim that some procedure will effectively recognize membership in A. Formally, productive sets are defined as follows.

Definition 3.1.4 Let W_i, $i \in \mathbb{N}$, be a numbering (i.e., a surjective assignment of natural numbers to the elements of a set of objects) of the recursively enumerable subsets of $\mathbb{N}$ such that $W_i = \operatorname{dom} \varphi_i$ (i.e., the domain of φ_i), where φ_i is a recursive function whose Gödel number is i. A set $A \subseteq \mathbb{N}$ is called productive if there exists a computable function ψ such that for all $i \in \mathbb{N}$, $W_i \subseteq A$ implies that $\psi(i)$ is *convergent* (i.e., it is defined) and $\psi(i) \in A \setminus W_i$.

There are productive sets in Π_1^0. For instance, the set N of all pairs $(\mathscr{M}, x)$ such that $\Psi_{\mathscr{M}}^{(1)}(x)$ fails to halt is a productive set that is in Π_1^0. In addition, there is a nonhalting procedure to determine whether a pair $(\mathscr{M}, x)$ belongs to N. And according to Kugel, the input processor employs such a nonhalting procedure to recognize objects.

The human mind is able to derive general theories from specific evidence and to deduce specific facts from its ever changing knowledge of the world. In general, theories are assumed to be correct until some evidence forces us to alter the theory or even to abandon it in favor of a new theory. For instance, as Kugel [104] has pointed out if all swans that we have observed are white, then we will come up with the theory that all swans are white. Naturally, this theory will change the very day someone observes a *cygnus atratus* (a black swan). This scenario of scientific research suggests that our ability to develop theories from specific evidence is not really computable (i.e., one cannot "re-create" this procedure by using a conventional computing device). Indeed, Kugel has suggested that this ability might be actually a trial-and-error procedure. As such, it can evaluate predicates in Σ_2^0. Thus, Kugel has actually suggested that the output processor is a Σ_2^0 process.

A simple model of how the mind actually solves problems is based on the solution of the *problem of inverting computable functions*, that is, given a machine p and an output o, find an input i such that $\Psi_{\mathscr{U}}^2(p, i)$ equals o. This problem can be solved by employing a k-limiting recursive process, and so one may say that the central processor is actually a k-limiting recursive

process, though it is not clear what the value of k shoulf be.

Usually, most computer programs decrease the amount of information involved during their execution. For example, a simple program that adds its (command line) arguments generates one number out of two. On the other hand, Kugel asserts that the selection of a program to perform a particular task increases the amount of information in an information-theoretic sense. This is an indication that this procedure cannot be computable. In general, a k-limiting recursive process can be used in association with a program-generating program to find a program that matches the evidence provided. This is clearly the task performed by the program selector. However, since many of the generated programs are not suitable for some particular task (e.g., they cannot handle all pieces of evidence), we need a mechanism to filter these programs. It is not possible to computably filter out all and only the totally computable programs from the list of all possible programs. But it is possible to perform this task noncomputably using a Π_2^0 filter, which explains why Kugel has suggested that the program selector is a Π_2^0 process.

The adoption of Kugel's model automatically implies the invalidation of the Church-Turing thesis. However, what is really puzzling about Kugel's model is that he asserts that most (if not all) vital mental processes are purely computational in nature. Obviously, a number of mental processes are indeed computational in nature, for instance, our ability to perform basic arithmetic operations.[3] However, it is one thing to be able to calculate the sum or the product of two numbers and another to fall in love and express it by saying "Sigga, I love you!" In other words, as has been already pointed out, no one has provided enough evidence to support the idea that feelings and affection are computational in their nature. Another aspect of Kugel's model is that it seems to be naive in the eyes of contemporary thinkers and researchers, for it lacks the "sophistication" of modern approaches to the philosophy of mind. Apart from this, it is really difficult to see why some machine that can solve the halting problem, can ipso facto feel angry, fall in love, or even worship God!

3.2 TAE-Computability

Jaakko Hintikka and Arto Mutanen [83, Chapter 9] present an alternative conceptual computing device that is similar to trial-and-error machines. The Hintikka-Mutanen abstract computing device is essentially a Turing machine with an extra tape, which is called the *bookkeeping* or *result-recording* tape. Both the working and bookkeeping tapes can be viewed as

3. Although it is not clear at all that our ability to perform basic arithmetic operations is computational, still, for the sake of argument, I will assume this is the case.

read-write storage devices, since the machine can print and erase information from either tape. Without loss of generality, one can assume that what appears on the bookkeeping tape are equations of the form $f(a) = b$, where $a, b \in \mathbb{N}$. These equations are used by the machine to define the function to be computed. In particular, this function is computed by the machine if and only if all (and only) such true equations appear on the bookkeeping tape when the machine has completed its operation. Practically, this means that each true equation $f(a) = b$ will be printed on the bookkeeping tape sometime during the operation of the machine and it will stay on it until the machine terminates. At that point, for each a, there has to be one and only one true equation on the bookkeeping tape. If no such equation has appeared on the bookkeeping tape for some a, or the equation for some a was changing continuously, then the value $f(a)$ is not defined.

A Hintikka-Mutanen machine cannot be simulated by a Turing machine, since these machines introduce a wider notion of computability compared to standard Turing machines. On the other hand, by imposing some restrictions on the operation of the machine, we obtain an abstract machine that is computationally equivalent to the Turing machine. More specifically, if we require that the machine never erase anything from the bookkeeping tape, the machine will behave like an ordinary Turing machine. In classical computability it is not enough to have each true equation $f(a) = b$ on the result-recording tape from some finite stage on, but it is necessary to know when the machine has reached this stage. If we allow the machine to erase data from the bookkeeping tape, then we could specify as a condition that $f(x)$ have the value b if and only if the equation $f(a) = b$ is the last equation of the form $f(a) = x$ produced by the machine. Clearly, this implies that we have at our disposal an effective procedure to determine when the last equation has been printed on the bookkeeping tape.

Hintikka and Mutanen call the resulting computability theory *TAE-computability*, short for trial-and-error computability. The following passage [83] gives an explanation of why this particular name was chosen:

> The name is motivated by the fact that erasure from the result tape is permitted by our definition. Such erasure can be thought of as an acknowledgement on the part of the machine that its trial choice of a line of computation has been in error and that it is using the recognition of an error to try a different line of computation.

TAE-computability is "arguably more fundamental theoretically than recursivity." In order to show this, the authors prove a theorem. But in order to fully comprehend it, one must be familiar with a number of definitions from logic. So, I will briefly present the notions of satisfiable formulas and Skolem functions as they are presented in [53]. Readers familiar with these concepts can safely skip the next paragraph.

Assume that L is a language (i.e., a countable set of nonlogical symbols). Then an *interpretation I of L* is characterized by the following:

(i) There is a *domain* of interpretation, which is a nonempty set D of *values*.

(ii) A function $f_I : D^n \to D$ is assigned to each n-ary function symbol $f \in L$. Constants are assigned values from D.

(iii) A proposition letter in L is assigned either the value tt or ff.

(iv) A relation $P_I \subseteq D^n$ is assigned to each n-ary predicate symbol $P \in L$.

Suppose that L is a language, I an interpretation of L, D the domain of I, and α a formula of L. Now, if α has the value tt in I for every assignment of the values of D to the free variables of α, then α is said to be valid in I or that I *satisfies* α. For any formula that is valid in an interpretation I, the interpretation I is called a model of α. Also, any formula α that has at least one model is called satisfiable, or else it is called unsatisfiable. A sentence is a logic formula in which every variable is quantified. A sentence is in prenex normal form if it has the following form:

$$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \alpha,$$

where Q_i is either a universal quantifier or an existential quantifier, x_i are distinct variables and each of them occurs at least once in α, and α contains no quantifiers. If σ is a sentence in prenex normal form, the Skolemization of σ is the procedure by which we eliminate each existential quantifier and its attached variable from the prefix of σ and then replace each occurrence of the attached variable in the quantifier-free part of the sentence with certain terms called Skolem functions. If the existential quantifier is in the scope of a sequence of universal quantifiers, then each free occurrence of the attached variable will be replaced by the term $f(x_1, x_2, \ldots, x_n)$, where f is a fresh function symbol and the x_i's are the variables attached to the universal quantifiers; otherwise, each free occurrence of the attached variable will be replaced by a new constant symbol.

In the 9th chapter of [83], Hintikka and Mutanen state and prove the following theorem.

Theorem 3.2.1 *Each satisfiable formula S of first-order logic has at least one model where its Skolem functions are TAE-computable.*

The essence of this theorem is that since there are satisfiable formulas for which there are no models with recursively enumerable relations (i.e., there are no sets of recursive Skolem functions) and we, on the other hand, can compute these sets using TAE-machines, these machines are clearly more powerful than Turing machines. Thus TAE-machines are hypermachines.

It is rather interesting to note that Hintikka and Mutanen conclude that the logic associated with TAE-computability is just classical logic. Going one (probably arbitrary) step ahead, one may say that classical logic is the logic of (one form of) hypercomputation.

Although TAE-computability is not as mature a theory as its classical counterpart, still there are certain aspects of the theory that have been addressed by its developers. For instance, a set is *TAE-enumerable* if and only if its *semicharacteristic* function[4] is TAE-computable. Also, a set is TAE-decidable if and only if both it and its complement are TAE-enumerable. The halting problem in the case of TAE-computability is formulated as follows: does the Turing machine with number n, which defines a partial function $f_n(x)$, TAE-compute a value for the argument m? It important to say that in the case of the TAE-"halting" problem we are not really concerned whether the machine will actually stop; instead, we are concerned about the constancy of the value the machine has reached.

For reasons of completeness, we present some results from [83, pp. 183–184].

Theorem 3.2.2 *If the sets A and B are TAE-enumerable, then so are $A \cap B$ and $A \cup B$.*

Theorem 3.2.3 *If the functions f and g are TAE-computable, then so is $f \circ g$.*

Theorem 3.2.4 *Being TAE-computable is an arithmetic predicate. In fact, it is a Σ_2^0 predicate.*

There are a number of interesting philosophical issues that are addressed by Hintikka and Mutanen. However, I will not discuss them here. The interested reader should consult [83] for more details.

3.3 Inductive Turing Machines

Inductive Turing machines were introduced by Mark Burgin and are described in detail in his recent monograph [28]. Generally speaking, a simple inductive machine is a Turing machine equipped with two additional tapes, each having its own scanning head. Burgin argues that the structure of a simple inductive Turing machine closely resembles the generalized architecture of modern computers. For instance, the input tape corresponds to the input devices of the computer (e.g., the keyboard, the mouse, the optical scanner), the output tape corresponds to the output devices of the

4. Given a set $A \subseteq X$, where X is some universe set, then for any $x \in A$, we have $c_A(x) = \chi_A(x)$, where c_A is the semicharacteristic function of A. When $x \notin A$, then $c_A(x) = \perp$, where $\perp$ denotes the undefined value. In other words, $c_A(x)$ is undefined when $x \notin A$.

computer (e.g., the video monitor, the printer), and the working tape corresponds to the central processing unit of the computer.

A simple inductive machine operates in a fashion similar to that of an ordinary Turing machine (e.g., the scanning heads read the symbol that is printed on a particular cell on the corresponding tape, then the machine consults the controlling device, and proceeds accordingly). However, their difference lies in the way they determine their outputs (i.e., the result of the computation). In the course of its operation, an inductive machine prints symbols on consecutive cells, which form sequences of symbols that form the result of the computation (Burgin calls these sequences *words*, but I prefer the term strings). Sometimes, the machine stops, provided it has entered its halting state, and thus operates like a normal Turing machine. Nevertheless, there are cases in which the machine does not actually stop. But this does not prevent the machine from giving results. When the machine has printed a string on the output tape that remains unchanged while the machine continues its operation, we can safely assume that this particular string is the result of the computation. Even in cases in which the result changes occasionally, it is quite possible that the output is adequate for our purposes. For example, when we compute a real number we are interested in computing it to a specific accuracy. Thus, when our machine has achieved computing the real number to the desired accuracy, we can fetch our result while the machine continues computing the number to even greater accuracy.

One can easily prove the following statement concerning the computational power of simple inductive machines.

Theorem 3.3.1 *For any Turing machine $\mathscr{T}$, there is an inductive Turing machine $\mathscr{M}$ such that $\mathscr{M}$ computes the same function as $\mathscr{T}$; that is, $\mathscr{M}$ and $\mathscr{T}$ are functionally equivalent.*

In order to classify simple inductive machines as hypermachines, they should be able to compute functions that ordinary Turing machines fail to compute. Clearly, in most cases we are interested in seeing how a potential hypermachine can solve the halting problem. Here is how this can be done: Assume that $\mathscr{M}$ is an inductive machine that contains a universal Turing machine $\mathscr{U}$ as a subroutine. Given a string u and a description $D(\mathscr{T})$ of a Turing machine $\mathscr{T}$, machine $\mathscr{M}$ uses machine $\mathscr{U}$ to simulate $\mathscr{T}$ with input u. In the course of its operation $\mathscr{M}$ prints a zero on the output tape. If $\mathscr{U}$ stops, which means that $\mathscr{T}$ halts with input u, machine $\mathscr{M}$ prints a 1 on the output tape. Now, according to the definition, the computational result of $\mathscr{M}$ is equal to 1 if $\mathscr{T}$ halts, or else it is equal to 0.

As has been demonstrated, simple inductive machines are hypermachines. However, the crucial question is, how much more powerful than ordinary Turing machines are these machines? It has been shown that these machines can compute functions that are in $\Sigma_3^0 \cap \Pi_3^0$, which is not really high

in the arithmetic hierarchy. For this reason, Burgin has developed an advanced form of inductive machine called *inductive Turing machines with a structured memory*. We note that these machines were developed independently from the theory of limiting recursion. For reasons of brevity, in what follows, the term "inductive machine" will refer to inductive machines with a structured memory.

A typical inductive machine consists of three components: *hardware*, *software*, and *infware*. The term infware refers to the data processed by the machine. An inductive machine $\mathscr{M}$ operates on strings of a formal language. In other words, the formal languages with which $\mathscr{M}$ works constitute its infware. Usually, these languages are divided into three categories: input, output, and working language(s). Normally, a formal language L is defined by an alphabet (i.e., a set of symbols on which this language is built) and formation rules (i.e., rules that specify which strings count as well-formed). The language L of an inductive machine is a triple (L_i, L_w, L_o), where L_i is the input language, L_w is the working language, and L_o is the output language. Notice that in the most general case it holds that $L_i \neq L_w \neq L_o \neq L_i$.

The hardware of an inductive machine is simply its control device, which controls the operation of the machine; its operating devices, which correspond to one or several scanning heads of an ordinary Turing machine; and its memory, which corresponds to one or several tapes of an ordinary Turing machine. The control device has a configuration $S = (q_0, Q, F)$, where Q is the set of states, $q_0 \in Q$ is called the initial state, and $F \subseteq Q$ is the set of final (or accepting) states. The memory is divided into different, but usually uniform, cells. In addition, it is structured by a system of mathematical relations that establish ties between cells. On each cell the operating device may print any of the symbols of the alphabet or it may erase the symbol that is printed on the cell. Formally, the memory is a triad $E = (P, W, K)$, where P is the set of all cells, W is the set of connection types, and $K \subseteq P \times P$ is the binary relation that specifies the ties between cells. Moreover, the set P, and consequently the relation K, may be a set with structure. A type is assigned to each tie from K by the mapping $\tau : K \to W$.

In general, the cells of the memory may have different types. This classification is represented by the mapping $\iota : P \to V$, where V is the set of cell types. Clearly, different types of cells may be used to store different kinds of information, but we will not elaborate on this issue.

The set of cells P is actually the union of three disjoint sets P_i, P_w, and P_o, where P_i is the set of input registers, P_w is the working memory, and P_o is the set of output registers. In addition, K is the union of three disjoint sets K_i, K_w, and K_o that define ties between the cells from P_i, P_w, and P_o, respectively. For simplicity, one may consider P_i and P_o to be two different singleton sets (i.e., to correspond to two different one-dimensional tapes).

The software of an inductive machine is a sequence of simple rewriting

rules of the following form:

$$\begin{aligned} q_h a_i &\longrightarrow a_j q_k, \\ q_h a_i &\longrightarrow C(l) q_k. \end{aligned}$$

It is also possible to use only rules of one form,

$$q_h a_i \longrightarrow a_j q_k c.$$

Here q_h and q_k are states of the control device, a_i and a_j are symbols of the alphabet of the machine, and c is a type of connection from K. Each rule instructs the inductive machine to execute one step of computation. For example, the meaning of the first rule is that if the control device is in state q_h and the operating device has scanned the symbol a_i, then the control device enters state q_k and the operating device prints the symbol a_j on the current cell and moves to the next cell. The third rule is the same except that the operating device uses a connection of type c, and in the case of the second rule, the operating device moves to the cell with number l. Having described in a nutshell the structure of inductive machines as well as the way they operate, we can now proceed with the presentation of results concerning the computational power of inductive machines.

First of all, let us see whether it is ever necessary for an inductive machine to stop and give a result. The following statement gives a negative response to this requirement.

Lemma 3.3.1 *For any inductive machine $\mathcal{M}$, there is an inductive machine $\mathcal{G}$ such that $\mathcal{G}$ never stops and computes the same functions as $\mathcal{M}$; that is, $\mathcal{M}$ and $\mathcal{G}$ are functionally the same.*

Also, the following result is quite important.

Theorem 3.3.2 *For any Turing machine $\mathcal{T}$ with an advice function (see Section 5.4), there exists an inductive Turing machine $\mathcal{M}$ with a structured memory that computes the same function as $\mathcal{T}$.*

In order to present the next result we need a few auxiliary definitions.

Definition 3.3.1 The memory E of an inductive machine is called recursive if the relation $K \subseteq P \times P$ and all mappings $\tau : K \to W$ and $\iota : P \to V$ are recursive.

The following result is not the one promised above. Nevertheless, it is a useful one.

Theorem 3.3.3 *An inductive machine with recursive memory is equivalent to a simple inductive machine.*

Definition 3.3.2 The memory E of an inductive machine $\mathscr{M}$ is called 0-inductive if it is recursive. For every $n \geq 1$, an inductive machine $\mathscr{M}$ with structured memory E is said to be $(n-1)$-inductive when the relation $K \subseteq P \times P$ and all mappings $\tau : K \to W$ and $\iota : P \to V$ are defined by some inductive machines of order n.

And here is the main result.

Theorem 3.3.4 *For any arithmetic relation Y, there exists an inductive machine $\mathscr{M}$ such that it computes the characteristic function of Y. If $Y \in \Sigma_n^0 \cup \Pi_n^0$, there is an inductive machine $\mathscr{M}$ of order n that decides Y.*

It is important to note that inductive machines are not only more powerful than Turing machines, but also more efficient. In addition, it can be shown that for a model of computation based on recursive functions, it is possible to find a class of inductive machines that can compute the same result more efficiently (personal communication with Burgin, 2005). Roughly speaking, the term efficiency means that computations performed by inductive machines take less time than their Turing counterparts. Also, when an inductive machine has delivered its result, it does not necessarily stop but can continue to operate, or as Burgin has put it in a personal communication:

> They always finish computation in a finite number of steps when they give the result, but they can continue to function. For example, when you wrote your e-mail to me, you gave a result, but I hope that you did not terminate your functioning.

3.4 Extensions to the Standard Model of Computation

When Turing proposed the abstract computing device that bears his name, he also proposed two other conceptual devices that somehow extend the capabilities of the standard Turing machine. These conceptual devices were dubbed choice and oracle Turing machines. Here is how Turing defined choice machines [206]:

> For some purposes we might use machines (choice machines or c-machines) whose motion is only partially determined by the configuration (hence the use of the word "possible" in §1). When such a machine reaches one of these ambiguous configurations, it cannot go on until some arbitrary choice has been made by an external operator.

The external operator is supposed to be a human being that assists the machine in the course of its operation. Clearly, if the actions of the mind transcend the capabilities of the standard model of computation, then c-machines are hypermachines by definition. It is rather interesting to note that a c-machine cannot be mimicked by a nondeterministic Turing machine (see Appendix A), since nondeterminism does not confer additional computational power on a Turing machine.

As we have described on page 12, an oracle machine is equipped with an external agency that can give correct answers to questions about a set $A \subset \mathbb{N}$. Clearly, it is possible to posit the existence of an oracle (i.e., a physical oracle) that gives correct answers to questions about a noncomputable set B. In fact, this is how Copeland has interpreted Turing's writing. On the other hand, no one has ever formulated oracles this way. Obviously, a machine assisted by such an oracle can compute sets and functions that are classically noncomputable. For example, a physical oracle machine might solve the halting problem for ordinary Turing machines. Naturally, for Copeland the next step was to propose oracle machines as a model of the human mind [34]:

> As I argued in my [previous] paper, O-machines point up the fact that the concept of a programmed machine whose activity consists of the manipulation of formal symbols is *more general* than the restricted notion of formal symbol-manipulation targeted in the Chinese room argument. The Chinese room argument depends upon the occupant of the room–a human clerk working by rote and unaided by machinery; call him or her Clerk–being able to carry out by hand each operation that the program in question calls for (or in one version of the argument, to carry them out in his or her head). Yet an O-machine's program may call for fundamental symbol-manipulation processes that rote-worker Clerk is incapable of carrying out. In such a case, there is no possibility of Searle's Chinese room argument being deployed successfully against the functionalist hypothesis that the brain instantiates an O-machine–a hypothesis which Searle will presumably find as "antibiological" as other functionalisms.

However, Bringsjord, Paul Bello, and David Ferrucci totally disagree with this idea. In particular, these authors point out that oracle Turing machines process symbols just like ordinary Turing machines [24]. In other words, Copeland's argument *falls prey* to Searle's argument. After all, one can supply a Turing machine with an auxiliary infinite tape (instead of a physical oracle) on which are listed, in increasing order (as sequences of 1's) the members of some set X. These machines can correctly answer any question regarding X and thus have the computational power of oracle machines. Obviously, these machines can be used to refute Copeland's

argument, since they are clearly symbol-manipulation devices. An interesting question is what happens when there is no auxiliary infinite tape, but a physical oracle, which leads naturally to the next question: do there exist physical oracles?

On page 15 we presented an alternative formulation of classical computability in the form of a random-access machine. An oracle Turing machine can be "simulated" by introducing a **read** command:

read `variable`

This command is an ordinary input command–nothing magical is assumed! However, the command is used only to assist the computation via an external operator (a physical oracle?), much as interactive systems take user feedback to proceed.

If we go one step further and introduce an *output* command (e.g., a **write** command) that can feed the "external world" with data, then we have a model of interactive computation. However, this model of computation is not general enough, since it suffers from the same drawbacks the classical model does. Nevertheless, it seems to be a step forward.

Coupled Turing machines, which were proposed by Copeland [38], are an extension of the notion of a Turing machine that exhibits interactive behavior. A coupled Turing machine is the result of coupling a Turing machine to its environment via one or more input channels. Each channel supplies a stream of symbols to the tape as the machine operates. In addition, the machine may also have one or more output channels that output symbols to the environment. The universal Turing machine is not always able to simulate a coupled Turing machine that never halts (think of a computer operating system, which is a system that never halts; nevertheless, sometimes some "operating systems" crash quite unexpectedly...).

It is not difficult to see that coupled Turing machines are actually hypermachines. Assume that C is a coupled Turing machine with a single input channel. The number of output channels will not concern us here. Also, suppose that $u \in [0, 1]$ is some "noncomputable" real number (i.e., a number that cannot be computed by a Turing machine) whose decimal representation can be written as follows: $0.u_1u_2u_3\ldots$. The digits of the binary representation of u will form the input of C. The input channel of C writes to a single square of the machine's tape, and each successive symbol u_i in the input stream overwrites its predecessor on this square. As each input symbol arrives, C performs some elementary computation (e.g., it multiplies the symbol by 3) and writes the result on some designated squares of the tape. In order to achieve constant operation time, the next result always overwrites its predecessor. No Turing machine can produce the sequence $3 \cdot u_1$, $3 \cdot u_2$, etc. (for if it could, it could also be in the process of producing the binary representation of u).

Clearly, the important question is what numbers a coupled Turing machine can compute. To say that it can compute more than the Turing

machine is not really useful. In addition, the vague description above is surely not a replacement for a rigorous mathematical definition of the machine and its semantics.

3.5 Exotic Machines

The term "exotic machines" refers to conceptual computing devices that assume that our universe has certain properties. For example, take the case of Thomson's lamp, which was "invented" by James Thomson and was first described in [202]. This is a device that consists of a lamp and an electrical switch that can be used to turn the lamp on and off. Assume that at $t = 0$, the lamp is off. At time $t = \frac{1}{2}$, we turn the lamp on. At time $t = \frac{1}{2} + \frac{1}{4}$, we turn the lamp off. At time $t = \frac{1}{2} + \frac{1}{4} + \frac{1}{8}$, we turn the lamp on. At time $t = \frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}$, we turn the lamp off and so on. The problem is to determine whether the lamp will be on or off at time $t = 1$. Thomson provided the following solution to this problem: assume that $0 < t < 1$. (i) If the lamp is off at t, then there is a t' such that $t < t' < 1$ and the lamp is on at t', and (ii) if the lamp is on at t, then there is a t' such that $t < t' < 1$ and the lamp is off at t'. Thomson thought that it followed from (i) that the lamp cannot be off at $t = 1$ and from (ii) that the lamp cannot be on at $t = 1$. This is clearly a contradiction, and thus Thomson concluded that this device is logically or conceptually impossible. Paul Benacerraf [12] has pointed out the fallaciousness of this argument. He claimed that one should distinguish between the series of instants of time in which the actions of the *supertask*[5] are performed (which will be called the t-series) and the instant $t^* = 1$, the first instant after the supertask.

Thesis 3.5.1 *From a description of the t-series, nothing follows about any point outside the t-series.*

From a practical point of view, one may say that tasks like this are really meaningless if time is granular. However, if time and space are continuous, then this task has at least some physical basis (for more details, see the short discussion at the end of Section 8.3).

The so-called Zeus machine is an example of an exotic machine that has been popularized by Boolos and Jeffrey in their classical textbook [18]. A Zeus machine is operated by the superhuman being Zeus (i.e., the principal god of the ancient Greek pantheon), who can perform an infinite task in a finite amount of time. Actually, Zeus can enumerate the elements of an

5. In philosophy, a supertask is a task involving an infinite number of steps, completed in a finite amount of time. The term supertask was coined by James Thomson.

enumerable set[6] in one second by writing out an infinite list faster and faster. In particular, Zeus enumerates the elements of the set in a way that is identical to the operation of Thomson's lamp. Copeland has proposed a more formal version of a Zeus machine, which is called an *accelerating Turing machine* [34]. These are Turing machines that perform the second primitive operation in half the time taken to perform the first, the third in half the time taken to perform the second, and so on. If we assume that the first primitive operation is executed in one minute, then since

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n} + \frac{1}{2^{n+1}} + \cdots < 1,$$

an accelerating Turing machine can execute infinitely many primitive operations before one minute of operating time has elapsed. It is interesting to see how accelerating Turing machines can compute the halting function. We assume that a universal accelerating Turing machine is equipped with a signaling device (e.g., a horn) that is used to send a signal when a computation is finished within one minute. In particular, given a Turing machine $\mathscr{M}$ with a Gödel number m that is supposed to compute the function $f(x)$, a universal accelerating Turing machine will take as input the numbers m and n (a possible argument to function f). If within one minute, the signaling device does not send a signal, the computation does not halt; otherwise, it does halt. Strictly speaking, this universal accelerating Turing machine is not a Turing machine at all, since it communicates with the external world. However, it is not really important to get into these details (the interested reader should consult Copeland's paper). Copeland claims that accelerating oracle Turing machines can be used to refute Searle's Chinese room argument. Again, this is not correct. Bringsjord, Bello, and Ferrucci [24] point out that

> After all, Zeus could be a pigeon. And a pigeon trained to move symbols around, even if blessed with the ability to carry out this movement at Zeus-level speeds, would still have the mental life of a bird, which of course falls far short of truly understanding Chinese.

Copeland responded to this argument by claiming that there is an *ascending hierarchy of symbol-manipulations* [35]. Thus, it is not possible to apply the Chinese room argument to all different levels of this hierarchy. However, symbol-manipulation is always the same kind of operation no matter how fast we perform it. Also, there are no recipes to construct a proof. Of course a brute-force search is not such a recipe, although it is employed by automated theorem-proving systems to prove truly interesting statements.

6. Although in the original description, Zeus is supposed to enumerate the elements of an enumerable set, it was pointed out to the author that a machine cannot so "easily" produce an enumeration of a countably infinite set. On the other hand, Zeus's job would make sense for any infinite recursively enumerable set.

But this approach is not always applicable. In addition, the general proof methodologies cannot be used to construct the proof or disproof of a particular mathematical statement. Also, mechanical symbol manipulation is a process that clearly lacks intentionality, and as Dale Jacquette remarks [90, p. 10]: "[T]he machine can only imperfectly simulate the mind's intentionality and understanding of a sentence's meaning." Now, whether machines have or do not have mental capabilities is an issue that I will address in Chapter 6.

The *Rapidly Accelerating Computer* (RAC), which was proposed by Ian Stewart [195], is actually equivalent to an accelerating Turing machine. In particular, the clock of an RAC accelerates exponentially fast with pulses at times $1 - 2^{-n}$ as $n \to \infty$. And just like accelerating Turing machines, an RAC can perform an infinite number of computations in a single minute. It can therefore solve the halting problem for Turing machines by running a computation in accelerating time and throwing a particular switch if and only if the Turing machine halts. Like all computations carried out by an RAC, the entire procedure is completed within one minute; and it siffices to inspect the switch to see whether it has been thrown. In Stewart's own words, "RAC can calculate the *incalculable*" (emphasis added). Interestingly enough, the RAC and accelerating Turing machines can be modeled by a classical (i.e., nonquantum) dynamical system, because classical mechanics poses no upper bound on velocities. Thus, it is possible to "accelerate" time so that infinite "subjective" time passes within a finite period of "objective" time. However, it is quite possible to achieve the same effect in special spacetimes, and we will say more on this matter on Chapter 8.

3.6 On Pseudorecursiveness

While working on his doctoral dissertation, Benjamin Wells constructed a particular nonrecursive set of algebraic equations, that his thesis advisor, Alfred Tarski declared to be decidable.[7] Thus Wells constructed a nonrecursive but decidable set. Clearly, the very existence of such a set jeopardizes the foundations of classical computability theory, which is of interest because it was constructed by someone without a negative attitude toward the Church–Turing thesis and its implications. Let us now see how one can construct such a set (the discussion that follows is based on Wells's two recent papers [224, 225]).

We want to construct a set of formal equations. Each formal equation

7. Any decision problem P is associated with a predicate F_P. In normal parlance, a problem P is decidable if F_P is computable; P is semidecidable if F_P is semicomputable; and P is cosemidecidable if $\neg F_P$ is semicomputable.

is a string that can be generated using the following formal grammar:

formal equation ::= term "=" term
term ::= "(" term + term ")" | constant | variable
constant ::= "a" | "b"
variable ::= "v_1" | "v_2" | $\cdots$ | "v_i" | $\cdots$ $i \in \mathbb{N}$.

We usually drop the outermost parentheses for clarity.

An *equational theory* of such formal equations is a set T consisting of strings, generated by the grammar above, that necessarily includes the equation $v_1 = v_1$. In addition, T must be closed under the following two operations:[8]

Subterm replacement The replacement of a subterm t_1 that appears in an equation in T by a term t_2 when $t_1 = t_2$ or $t_2 = t_1$ belongs to T.

Variable substitution The substitution of a chosen but arbitrary term for every occurrence of a variable in an equation in T belongs to T.

A subset B of an equational theory T is an *equational base* for T if T is the smallest equational theory that includes B. We write $T = \mathrm{Th}(B)$. Thus, T is *recursively based* precisely when T is the closure under subterm replacement and variable substitution of a finite set, or an infinite recursive set, of equations. The class of algebraic models for an equational theory is called its *variety*. The equational theories that Wells considered in his work are equational theories for varieties of *semigroups*, that is, they contain equations that guarantee the associativity of the + operator. In other words, they contain

$$(v_1 + v_2) + v_3 = v_1 + (v_2 + v_3).$$

In addition, one can introduce an *additive identity* element by including the following equations:

$$0 + v_1 = v_1 = v_1 + 0.$$

Here 0 is a new distinguished constant or a particular term.

Assume that T_n is the subset of the equational theory T consisting of the equations in T with no more than n distinct variables. We say that T is *quasirecursive* if for every number n, T_n is recursive. The theory T is *pseudorecursive* if T is quasirecursive but not recursive.

Wells has constructed various pseudorecursive equational theories. In particular, he provides the following recipe for constructing such theories: start with a fixed but arbitrary nonrecursive, recursively enumerable set $X \subset \mathbb{N}$ and define a finite equational base $\Psi 1_X$ from a highly engineered Turing machine that accepts X [223]. The resulting theory $\mathrm{Th}(\Psi 1_X)$ is an

8. A set X is said to be closed under some operation or map L if L maps elements of X to elements of X.

equational theory of semigroups with identity and finitely many individual constants (the number of constants can be reduced to two or even to one or zero). This recipe can be formally summarized as follows.

Theorem 3.6.1 *For every nonrecursive recursively enumerable set X that is a subset of* $\mathbb{N}$, $\mathrm{Th}(\Psi 1_X)$ *is a finitely based pseudorecursive equational theory that is Turing-equivalent to X.*

According to Wells [224], Tarski has suggested that a basis for a decision procedure of T (i.e., a method for telling whether an arbitrary equation is in T) can be constructed as follows: For each value of n, there is a procedure for deciding T_n; n can be used to index a catalog of these procedures. Given an arbitrary equation, count the number of variables in it, and then use the catalog to locate the correct procedure and apply it. Using this basis, we can construct a decision procedure for a finitely based or even just recursively enumerable pseudorecursive theory. We employ an oracle Turing machine, whose query tape lists the values of the characteristic function for X. The keying is performed by counting variables to n. Indexing depends on oracular information: the Turing index for the template used to build the machine M_n, which is used to decide T_n, and the first n items from the oracle are sent to an internal foundry to be recast as a functional equivalent of M_n, and its Gödel number, now computable, is returned. The last step sends this number with n to a universal Turing machine that simulates M_n computing with input n and "allows nature to take its course." This oracle Turing machine can decide T_n by looking only at the first n cells on the tape, which lists the characteristic function f_n of $X_n = \{0, 1, \ldots, n\} \cap X$. In addition, since this finite function is recursive, we can incorporate it into the control mechanism of the oracle machine to form M_n^*, an ordinary Turing machine that decides T_n. Clearly, for every $n, f(n) = f_n(n)$, where f is the characteristic function of X (therefore, there is no way to recursively recapture X from the X_n or synthesize an ordinary Turing machine to decide T).

The work on pseudorecursiveness has revealed the following (see the abstract of [225]):

> The dilemma of a decidable but not recursive set presents an impasse to standard computability theory. One way to break the impasse is to predicate that the theory is computable–in other words, hypercomputation by definition.

This statement should not be taken as an indication that hypercomputation is actually an empty word. On the contrary, the theory can be made computable once the very notion of computability is extended. Wells believes that one should expand a theory only when "real" problems need solutions. For example, our inability to find a number that is the square root of minus one led mathematicians to *invent* the imaginary numbers. Thus, Wells expects a new expanded theory based on nonrecursive yet decidable sets.