# A Formalisation of Non-Finite Computation

**Lucian Wischik**
**Queens' College**

**Abstract.** Recent work in the field of relativitistic space-times suggests that it may be possible for a machine to perform an infinite number of operations in a finite time. Investigation of these machines has three motivations. First, because the machines may be physically possible, they have implications for the general question of what problems are theoretically solvable. Second, the physical laws that govern their operation give rise to a rich mathematical structure. Third, by investigating general relativistic computation, we get a clearer picture of properties peculiar to the special case of Turing computation. A mathematical formalisation of the operation of the machines is presented, and shown to correspond to their physical operation. It is proved that there is no satisfactory way to give a finite description for the machines. Although Gödel sentences exist if attention is restricted to a finite set of machines, further results [Ear94,Hog96] about the computational power of the machines and their equivalence to the Kleene arithmetical hierarchy are shown to depend upon arbitrary assumptions. This gives rise to a non-finite version of the Church-Turing thesis. The case of non-finite computation is used to arrive at an abstract principle of computation that is independent of physics.

**A dissertation submitted to the University of Cambridge towards the degree of Master of Philosophy**

**June 1997**

# CONTENTS



# PREFACE


I am very grateful to my brother, Damon Wischik, for many hours of fruitful discussion. Lemma 2.5 and theorem 3.7 are due to him. The rest of the dissertation, except where otherwise stated, is the product of my own original research. I would also like to thank my supervisor, Mark Hogarth.

# INTRODUCTION

It has been suggested that, within certain relativistic space-times, it may be possible for a machine to perform an infinite number of operations in a finite time. Because such machines would be very powerful, their possible existence would have epistemological implications regarding the solvability of problems. It is therefore important that their computational power be formally characterised. One class of such relativistic space-times are known as *Malament-Hogarth* space-times (henceforth abbreviated to MH), and the machines that operate in them are called MH machines.

In the first chapter a precise logical notation is developed to characterise MH machines. This is necessary because such terms as an 'infinite number' require formal definition. The notation provides essentially a formal version of the description of the machines given by Hogarth [Hog96, 3.12]. It will be seen in the second chapter that important results about the computational power of the machine become evident only through the careful structured analysis achieved with the notation. This dissertation therefore develops the necessary formal definitions and proofs which form an integral part of the argument. Every definition and theorem is stated formally, followed by a non-technical explanation.

The significant results of the second chapter can be summarised as follows. There are uncountably many different MH machines: that is, there are more machines than there are natural numbers (lemma 2.4). This is in contrast to conventional Turing machines, of which there are exactly as many as there are natural numbers. Because there are uncountably many different machines, it is impossible to have a system of finite description, a 'programming language', that is capable of describing all of them. There is therefore no Gödel sentence for the machines. That is, there is no sentence whose existence would indicate that the system was either incomplete or inconsistent.

Given the above result it will be impossible to arrive at general theorems concerning all MH machines. We examine the feasibility of restricting the field of attention just to possible systems of finite description, which cover some but not all of the machines. But lemma 2.5 demonstrates that there is no such system that is inherently satisfactory: any system that is proposed, could be improved upon by expanding it to describe even more machines.

We therefore consider entire families of systems of finite description. Definition 2.6 introduces three different families, each progressively smaller and more arbitrary. The first contains what we call *sequencing-complete* systems of description. If we limit our attention only to machines which can be described with a sequencing-complete system, then Gödel sentences for MH machines become possible.

The second family of systems of finite description, smaller than the first, contains what we call *simulation-complete* systems of description. Gödel's Incompleteness Theorem [Göd31, theorem IX; see also Del70, pp. 162- for a helpful introduction] depends upon the fact that the system he chose was simulation-complete. The *a priori* reasons for requiring the same property of MH machines are debatable. But by restricting our attention to simulation-complete systems, we obtain the *size-power theorem:* the larger the machine, the more powerful it is.

The third family of systems of finite description, smaller still, contains what we call *Turing-generable* systems of description. There appears to be no justification for considering such systems. However, it is shown that previous results [Ear94,Hog96] which relates machine sizes

**to the Kleene arithmetical hierarchy[1], implicitly assume such a system of description. In particular, the previous results state that, in order to decide the truth of an arithmetical proposition, a machine is required whose size is related to the number of quantifiers of that proposition.**

**There appears to be no clear reason for choosing any one family over the others. The question of which to adopt is left for further inquiry. But the attempt at classification serves the useful secondary purpose of clarifying the criteria whereby which systems of finite description are chosen for conventional Turing machines.**

**The investigation of MH machines has a further advantage. The description of the physical construction of the machines leads to the following underlying principle of computation: a computation is a sequence of instructions to be executed such that, at any time during the execution, there is a unique instruction that will be executed next. This abstract notion of computation leads to three different categories: finite computations, countable ordinal computations, and uncountable ordinal computations. Conventional Turing machines are finite. The MH machines described in this paper perform countable ordinal computations (theorem 3.3). And uncountable ordinal computations could never be completed in a finite time (theorem 3.5). These abstract computational properties are independent of any considerations of physics or space-time. Therefore the answer to the epistemological question raised at the start of the dissertation is the following: the problems that are solvable systematically, by machine, are precisely those that can be solved by MH machines. This would appear to be the** *a priori* **limit of computability.**

---

**[1] The Kleene arithmetical hierarchy has arithmetical sentences ranked by the number of quantifiers. Sentences of the form $\forall x$.P($x$) and $\exists x$.P($x$) come first, with P a** *primitive recursive predicate***. Next come sentences with two quantifiers: $\forall x.\exists y$.P($x$,$y$) and $\exists x.\forall y$.P($x$,$y$). And the hierarchy continues with sentences with progressively more quantifiers. For an introduction to primitive recursive predicates, see [Kle67]**

# CHAPTER 1. LOGICAL NOTATION FOR MH MACHINES

**A *machine* is something that starts in some *state* and executes *instructions* in order. Instructions take the machine from one state to another.**

**Many different formalisms have been presented to describe machines, such as Turing machines [Tur37], lambda calculus [Chu36] and register-machines [Min67]. Issues concerning the infinite nature of infinite computation are independent of the particular formalism used. We therefore do not review the conventional formalisms; instead several alternatives will be presented which are more suited to the problems at hand.**

**What we call a state depends on our formalism: it might be a Turing machine configuration (describing the contents of the tape and current mode of the machine), or a lambda expression, or the values of a collection of registers along with the index of the current execution step, etc. What we call an instruction also depends on our formalism: it might be a Turing machine state transition, or a lambda reduction, or a register-machine step, etc.**

**As an example, consider a machine whose state is represented by a single integer and which has two instructions, INC and DEC. The instruction INC: $x \to x+1$ maps any number to its successor; the instruction DEC: $x+1 \to x$ does the reverse. Another example is a Turing machine which has exactly one instruction, STEP, which performs one state transition. In different Turing machines the instruction behaves in different ways; but each machine has only the single instruction.**

**Let $I_X$ be the set of all instructions in some formalism X (where the formalism might be Turing machines, the lambda calculus, register machines, etc.) Let $S_X$ be the set of states. $I_X$ is a set of maps from $S_X$ onto itself. There are further restrictions upon which instructions are allowable, as detailed below.**

**The first restriction is that the set of states and the set of instructions is countable. All the above examples meet this condition. Consider, for example, the Turing machine. Its configuration can be represented by four natural numbers (the numbers representing the digits respectively to the left of the tape head, those under it, those to its right; and the current machine mode). So, the set $S_{TM}$ of Turing machine states satisfies the relation $S_{TM} \subseteq N \times N \times N \times N$ and is therefore countable. The singleton set of instructions {STEP} is clearly countable.**

**The second restriction is that the states and instructions must be what would reasonable be considered the basic building blocks of algorithms. This restriction is necessarily vague, but consideration of the following examples leads to the definition proposed below. Turing machines are generally considered to be reasonable building blocks. Also reasonable are lambda terms [Chu36]: here the set of states $S_\lambda$ is the set of valid lambda expressions, and the set $I_\lambda$ contains instructions for ALPHA, BETA and GAMMA reduction. An example of a set of instructions that is not reasonable is the set $I_{halting}$ which contains only the single instruction DOES-IT-HALT?. This instruction, when started in state N, ends in state 0 or 1 according to whether the machine characterised by the number N would halt. We consider this unreasonable because it does not seem sufficiently 'basic'.**

*Definition 1.1*. **Computational formalisms.**

**(i) A computational formalism X consists of a countable set $S_X$ of states, and a countable set $I_X$ of instructions where each instruction is a map from $S_X$ onto itself. The letter C will be used to denote instructions.**

(ii) **A computational formalism is *reasonable* if there is a Turing machine into which it can be embedded so that every instruction in the formalism corresponds to some finite sequence of instructions of that Turing machine. That is, there exists an injection $f:S_X \rightarrow S_{TM}$ from the states of that formalism to the states of some particular Turing machine such that, for every $C \in I_X$ and $A,B \in S_X$, if $C(A)=B$ then there is some number $n$ such that $B = f^{-1} \bullet STEP^n \bullet f(A)$.**

**This definition expresses the essence of the Church-Turing thesis [Tur37, appendix]: any function that can be computed with a reasonable formalism in a finite number of steps, can also be computed with a Turing machine (or with lambda reduction, or a register machine, etc.) in a finite number of steps.**

**The definition of computation above is one in which many different formalisms can be expressed straightforwardly, and does not restricts us to any one particular formalism. The generality of this definition was chosen primarily for ease of presentation of the theorems and proofs in this dissertation. It also emphasises the fact that the issue of non-finite computation is totally independent of choice of formalism. Two notes of caution are required. First, it may be that one formalism can compute in a small number of steps what would take many more steps in another: computations that take one step are thus essentially similar to those that take two, or three, or any finite number of steps. Second, at times in this dissertation a very simple set of instructions is chosen to illustrate some point, and this set of instructions may have to be augmented to prove the next point. This again is simply a question of ease of presentation: had the initial set of instructions been sufficiently general[2], then all the new instructions could simply have been written in terms of the old. Similarly, when the set of states is extended so that the machine can store an extra number (using $S_X \times N$ instead of $S_X$) the difference is also not essential.**

**It was assumed above that the set $S_X$ of machine states was countable. This is because Turing machines have countably many states; and it would not be possible to embed an uncountable formalism into a Turing machine. The restriction to a countable set of instructions is valid because (lemma 3.5) it would be unreasonable for a computation to execute more than a countable number of instructions. The possible suggestion that we should restrict attention just to a finite set of instructions, rather than a countably finite set, is discussed at the end of the following section.**

## Pictorial explanation of MH machines.

**We now move to a logical notation, and we show that the notation corresponds to the characterisations of MH machines given by [Ear94,Hog96]. The first part of the argument involves an intermediate diagrammatic representation, with the expectation that the links between Hogarth's description and the diagrams, and those between the diagrams and the logical notation, are clear and unobjectionable. It is not necessary to justify the physicality of the diagrams here; it is sufficient merely to demonstrate their equivalence to Hogarth's description, since that has already been justified physically.**

**As mentioned, each instruction takes the machine from one state to another state. Instructions are represented as rectangles; the state going in and coming out is represented with a thick line (see Fig. 1). A finite computation is simply a sequence of instructions in which each instruction changes the state, in order (see Fig. 2). Time increases in the upward direction. Every**

[2] **A set of instructions is *Turing-powerful* if it can simulate the operation of any given Turing machine.**

**instruction is executed at a distinct period in time. Instructions that are higher up the page are executed after those that are lower.**

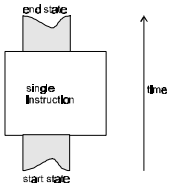


**Figure 1. A single machine instructions.**

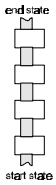


**Figure 2. A finite sequence of instructions.**

**Within MH space-times, it is possible to have a countably infinite *string* of instructions with a *solution event*, indicated by a filled dot, lying in the future of all of them. The terms 'string' and 'solution event' have precise technical meanings relating to the space-time layout of the machines [Hog96, 3.9]. For the purposes of this dissertation it is sufficient merely to note that the solution event comes after all instructions have been executed, and that there is a finite path λ from the start to the solution event (Fig. 3).**

**At each instruction it is possible for a signal to be sent to the solution event (Fig. 4). The question of the physical realisation of the signal can be set aside. It is sufficient to assume that there is some way of detecting the presence or absence of a signal, and that at the solution event it is possible to distinguish whether no signals have arrived, or whether at least one signal has arrived. The presence or absence of signals can be treated as a boolean values[3], with an infinite boolean OR operator applied to them all at the solution event. The machine pictured (Fig. 4), with an infinite string of instructions leading to a solution event, will be termed an ω-*machine*[4] (where ω is the conventional notation for the smallest countable infinity, and refers to the number of instructions in the machine).**

**The standard MH machine is taken to bifurcate at the start: one part goes through the countably infinite string of instructions, possibly sending signals to the solution event; the other part follows path λ directly to the solution event without changing its state at all[5]. It is possible to have further instructions after the solution event, and it is possible for these instructions to be influenced by whatever signals were sent to the solution event (Fig. 5). Earman and Norton [Ear94] call the *Master* that part that follows path λ, and the *Slave* the other.**

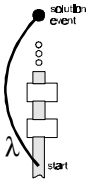


**Figure 3. An infinite string of instructions, with a solution event lying to the future of them. The path λ is finite.**

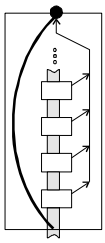

**Figure 4. In an infinite string, each instruction is able to send a signal to the solution event.**

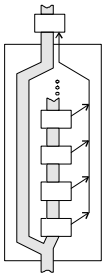

**Figure 5. An instruction after the solution event. Its behaviour may depend upon the signals it receives.**

---

**[3] A boolean value is either 0 or 1, for 'false' or 'true'.**

**[4] What we call an ω-machine is what Hogarth calls a $SAD_1$ space-time.**

**[5] Our treatment is slightly different to that of Hogarth. We consider that even the simplest infinite string requires a bifurcation so that there is a machine at the end of the string to deliver the result. Hogarth does not require a machine at the end. In his system it is only the more complicated machines that bifurcate.**

**The instructions as described so far have non-uniform properties. No instruction has an input signal apart from the one after the solution event. Every instruction inside the string has an output signal destined for a solution event but the instruction at the end does not. And the box containing the string has a single output signal destined for the following instruction, but nothing else has such an output signal. For convenience we will regularise the machines by making every instruction have an input signal, an output signal that is sent to the next instruction, an output signal that goes to some possible future solution event, and a state input and output (Figs. 6, 7). This lets us treat instructions, finite sequences and infinite sequences in exactly the same way.**

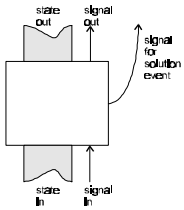


**Figure 6. A 'regularised' instruction, indicating possible routes for signals.**

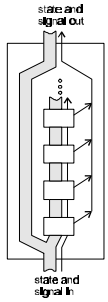


**Figure 7. How the signals connect when an infinite string of instructions leads up to a solution event.**

**It should be noted that the signals sent out by an instruction to some possible future solution event (Fig. 6) need not actually go anywhere. For example, in a machine which consists of only two instructions in sequence with no solution event, all such signals are simply ignored.**

**As mentioned above, the ω-machine consists of an infinite string of instructions leading to a solution event. It is further possible to construct a machine with an infinite string in which each box is not a single instruction, but an ω-machine [Hog96, def. 3.9.4]. The process may be repeated indefinitely: given any set of MH machines we can put them in an infinite string[6]. We define a *component* to be either a single instruction, or a finite sequence of two or more other components, or an infinite string of components leading to a solution event. An *MH-machine* is exactly the same as a component; we use the term 'component' when referring specifically to the construction of a machine.**

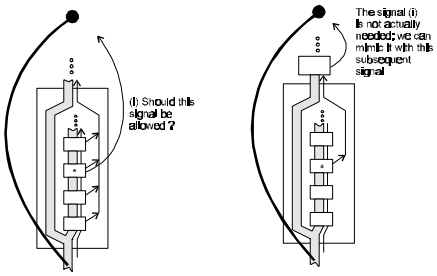


**Figure 8. Should instructions be allowed to send signals to many different solution events, different distances away? No: such flexibility is not needed.**

**[6] Hogarth considers [private communication] that the process of constructing a new machine from an infinite string of machines is always physically plausible.**

**In such an infinite string of ω-components, it might seem reasonable that the instructions within each ω-component should be able to send some signal out to solution events other than the immediate one (Fig. 8). This would require for example that the individual instruction marked with the asterisk be able to send out signals to three different destinations: to the next component, to an immediate solution event, or to a later solution event. But as shown in the diagram, this three-signal flexibility is not needed. The same result can be achieved with only two signals: one to the next component, and one to the immediate solution event[7].**

**It may seem arbitrary to restrict ourselves to only these two signals and not to some other number. The reason is essentially that in our definition of components there were only two constructive rules—finite sequencing and infinite sequencing—that could create larger components out of smaller. The signal sent to the next component corresponds to the finite sequencing rule, and the signal sent to the immediate solution event corresponds to the infinite sequencing rule. If we had presented further construction rules, then the machines would have needed more signals.**

**The diagrams presented above are essentially similar to those given by [Hog96, fig. 45]; but these make explicit the flow of information between the parts of the machine. Hogarth [cond. 3.12.1] imposed three additional conditions upon the flow of information, numbered (i) to (iii) below, to ensure the physicality of the machines. Earman and Norton [Ear94, p. 32] imposed two more, numbered (iv) and (v) below.**

> *Definition 1.2.* **Requirements of MH machines.**
> **(i)** *A lower box can signal to a higher box or to the solution event*. **In the diagrams we have given, lower boxes can signal only to the immediately following box or to the solution event. This is not a problematic restriction, however. If a box requires to send a signal to some subsequent box, the signal to the immediate next box is simply the request to 'pass the message along'.**
> **(ii)** *Each box and the signal event can receive at most one signal (no swamping)*. **This condition is not met in the above diagrams, since we have a countably infinite number of boolean signals arriving at the solution event which are assembled with an infinite OR operator. However, as will be proved (theorem 1.8), the difference is not important.**
> **(iii)** *Downward signalling is forbidden*, **because signals are not allowed to travel backwards in time. This condition is met simply because all arrows point upwards.**
> **(iv)** *The slave cannot leave a tape for inspection by the Master as output*. **That is, the state at the end of an infinite string cannot be left for the instruction after the solution event. This condition is met in the above diagrams by having the state at the solution event being simply a copy of the state at the start of the infinite string.**
> **(v)** *The Master may not infer results of computation by reading the limiting behaviour of an infinite sequence of signals emitted by the Slave in the course of the computation*. **This condition is met, since we take the infinite disjunction of boolean signals.**

**The possible objection has been raised that it is unreasonable to allow a countably infinite number of different instructions. However, consider for example the construction of machines given by Hogarth [Hog96, sec. 3.9]. He allows for an $\omega^2$ sized machine to be made up of an infinite string of ω-components, with each ω component containing any Turing machine: thus, he allows for an infinite number of different Turing machines. In definition 1.1 we defined that, in a reasonable computational formalism, every instruction is equivalent to a finite sequence of Turing machine transitions. Thus, allowing an infinite number of different Turing machines is effectively the same in our notation as allowing an infinite set of instructions.**

---

**[7] Strictly, for the two-signal machine to be equivalent to the three-signal machine, we would require two infinite-string components in sequence.**

**Additionally, while Hogarth allows any infinite string of components to make up more powerful machines, he only allows a single repeated instruction to constitute his ω-machine. The difference is a minor one, affecting only the smallest machines: both Hogarth's construction and the notation in this dissertation allow for arbitrary infinite strings at some level, and so both exhibit essentially the same behaviour. Since it seems arbitrary to impose the limitation only for ω-machines, and since its removal does not fundamentally affect the theorems in this dissertation, we will dispense with it.**

**We have established that the diagrams described in this chapter meet the restrictions upon machines imposed by [Ear94,Hog96]. The final task, of showing that the diagrams are at least as general, is essentially impossible. This is because the upper limits of a system can only be characterised adequately within a formalisation, whereas the machines presented in the literature to date have only informal descriptions. It is hoped that future discussion concerning which physical properties are possible and relevant will find a common frame of reference in the notation introduced in this dissertation[8].**

**This concludes the description of the construction of MH machines, and indicates that the diagrammatic construction presented above corresponds to the structure proposed by [Ear94,Hog96]. The next section describes more formally how the machines can be constructed out of a set $I_X$ of instructions. $M_X$ is the set of MH machines within the particular formalism X. The subscript X is generally omitted.**

## Logical notation for MH machines.

**In a conventional computation we write $I_X:S_X \to S_X$, meaning that instructions map from states to states. For MH machines, the instructions are augmented with signal information as per (Fig. 6) to give $I_X:S_X \times B \to S_X \times B \times B$, where $B$ is the set $\{0,1\}$ of booleans. If C is an instruction, then $C(S_0,b_0) = (S_1,b_1,u_0)$ means that the machine starts in state $S_0$ and receives a signal $b_0$ from the previous component; the execution of instruction C makes it go into state $S_1$, send the signal $b_1$ to the following component, and send the signal $u_0$ to the immediate solution event.**

> *Definition 1.2.* $M$**, the set of MH machines, is the smallest set generated by the three axioms of machine construction: a single instruction is a machine, any two machines in sequence are also a machine, and an infinite string of machines is also a machine. The axioms are presented as syntactic rules below. (The horizontal line represents a syntactic rewrite of a formula, from the premises on top to the conclusion underneath).**
>
> $$\textbf{(STEP-M)}\quad \frac{\mathbf{C} \in I}{\mathbf{C} \in M}$$
>
> $$\textbf{(SEQ-M)}\quad \frac{\mathbf{M_0} \in M \quad \mathbf{M_1} \in M}{\mathbf{M_0;M_1} \in M}$$
>
> $$\textbf{(MH-M)}\quad \frac{\forall i \in \omega.\ \mathbf{M_i} \in M}{\mathbf{MH}(\textstyle\sum_{i\in\omega}\mathbf{M_i}) \in M}$$
>
> **We sometimes write $\mathbf{MH}(\mathbf{M_X};\sum_{i\in\omega}\mathbf{M_i})$. This is a syntactic shorthand for $\mathbf{MH}(\sum_{i\in\omega}\mathbf{N_i})$, where $\mathbf{N_0}=\mathbf{M_X}$ and $\mathbf{N_{i+1}}=\mathbf{M_i}$.**

---

**[8] One particular suggestion [implied by Ear94] is that machines can send not just booleans as signals but also numbers. Machines which can send arbitrary numbers appear to be strictly more powerful than those that can send only booleans, since a number is equivalent to an arbitrarily long string of booleans. However, this remains to be proved.**

**The syntactic form of the final rule, $MH(\sum_{i\in\omega}M_i)$, has been chosen to indicate that the construction involves a solution event in an MH space-time and that it involves an infinite string sequenced together. However, the precise choice of symbols carries no formal meaning: we could equally have chosen a notation with just $MH_i(M_i)$, or $\sum_i M_i$.**

*Definition 1.3.* **The following rules describe the** *denotation* **of MH machines. The denotation [[M]] of a machine M is the set of instructions which it goes through, ordered in time. $C_0<C_1$ means that $C_0$ comes before $C_1$.**

**In a machine consisting of only a single instruction, the denotation is obvious. When two machines are sequenced together, the denotation consists of all the instructions in the first followed by all the instructions in the second. And the denotation of an infinite string of machines consists of all the instructions in each machine, in order. The mathematical term for all elements of one set followed by all elements of another is the** *ordered disjoint union* **of the sets.**

$$\text{(STEP-COMP)}\quad \frac{C\in I}{[[C]]=\{C\},\ C\leq C}$$

$$\text{(SEQ-COMP)}\quad \frac{[[M_0]]=(A_0,\leq_0)\quad [[M_1]]=(A_1,\leq_1)}{[[M_0;M_1]] = \{0\}\times[[M_0]] \cup \{1\}\times[[M_1]]}$$

$$\text{(MH-COMP)}\quad \frac{\forall i\in\omega.\ [[M_i]]=(A_i,\leq_i)}{[[\ MH(\sum_{i\in\omega}M_i)\ ]] = \cup_{i\in\omega}\ \{i\}\times[[M_i]]}$$

**The disjoint unions in SEQ-COMP and MH-COMP are ordered** *lexicographically***—that is, in dictionary order with the first element sorted before the second.**
**$(a,b)\leq(c,d) \Leftrightarrow a<c \vee (a=c \wedge b\leq d)$.**

**The** *type* **of a ordered set is a standard mathematical term referring to its size and ordering structure[9]. Lemma 3.5 proves that the denotation of MH machines has a** *countable ordinal type***.**

*Definition 1.5*. **Size of machines.**

**(i)** **The** *ordinal size***, or just** *size* **of a machine, written ord[[M]], is the ordinal type of [[M]].**

**(ii)** **A** *limit machine* **is one whose size is a limit ordinal [Hal60, p.113]. This implies that it has a solution event at its end. The ω-machine (an infinite string of instructions leading to a solution event) is a limit machine; the machine composed of two instructions in sequence is not. Limit machines will be used in some proofs in chapter two because their size is invariant under Church-Turing equivalence. That is, the restriction mentioned at the start of the chapter—that machines which perform one instruction are equivalent to those that perform two, or three, or any finite number—becomes irrelevant since limit machines contain no finite sequence of instructions.**

**Hogarth [Hog96, fig. 45] presents an axiom schema consisting of three rules, listed below, for the construction MH machines. The axioms have been rewritten in the notation introduced in this chapter. The precise behaviour of the instruction STEP in the machines is described by Hogarth, but since we are interested for the moment in the size of the machines its behaviour is unimportant.**

---

**[9] The treatment of ordinals in this dissertation is closest to that given by [Hal60]. A concise introduction is [Joh3].**

(i) The definition $SAD_1 \equiv MH(\sum_i STEP)$

(ii) A rule for generating $SAD_{n+1}$ from $SAD_n$: $SAD_{n+1} \equiv MH(\sum_i SAD_n)$.

(iii) The definition $AD \equiv MH(\sum_i SAD_i)$.

The ordinal sizes of these machines are $ord[[SAD_1]]=\omega$, $ord[[SAD_{n+1}]]=ord[[SAD_n]]\times\omega=\omega^{n+1}$, and $ord[[AD]]=\omega^\omega$. Effectively, axiom (ii) corresponds to ordinal multiplication and axiom (iii) corresponds to ordinal exponentiation; equivalently, we could say that axiom (iii) represents $\omega$ applications of multiplication. The next logical axiom would be for further exponentiation, giving

(iv) $AD_{n+1} \equiv MH(\sum (MH(\sum \ldots AD_n) ) )$, with $\omega$ applications of the MH rule.

This would give $ord[[AD_{n+1}]]=ord[[AD_n]]^\omega=((\omega^\omega)^\omega)\ldots$, with n exponentiations. Then it would be necessary to invent a further axiom for $\omega$ applications of exponentiation, and so on with a countably infinite number of axioms. This notation, which requires explicit rules for the construction of each machine, rapidly becomes unwieldy. The rules introduced in this dissertation (definition 1.2) do not suffer from the same shortcoming, because they do not label each individual machine.

## Execution of machines

The notation $[S_0,b_0]M[S_1,b_1,u_0]$ will be used to mean that, having been started in state $S_0$ with a signal $b_0$, the machine will end in state $S_1$, send a signal $b_1$ to the next component, and send the signal $u_0$ to some possible future solution event if there is one.

*Definition 1.5.* **The following rules describe the effect of execution of MH machines, and are essentially a syntactic form of the diagrams presented earlier in the chapter. Note that, in an infinite string of components, an infinite boolean OR operator is applied to all of the signals sent out to the solution event. The result of the operator is taken as the signal sent by the infinite string to its next component; the infinite string itself sends no signal out to any possible future event (Fig. 8).**

$$\text{(STEP-TRAN)}\quad \frac{C \in I_X \quad C(S_0,b_0)=(S_1,b_1,u_0)}{[S_0,b_0]\ C\ [S_1,b_1,u_0]}$$

$$\text{(SEQ-TRAN)}\quad \frac{[S_0,b_0]\ M_0\ [S_1,b_1,u_0] \quad [S_1,b_1]\ M_1\ [S_2,b_2,u_1]}{[S_0,b_0]\ M_0;M_1\ [S_2,b_2,u_0 \vee u_1]}$$

$$\text{(MH-TRAN)}\quad \frac{\forall i \in \omega.\quad [S_i,b_i]\ M_i\ [S_{i+1},b_{i+1},u_i]}{[S_0,b_0]\ MH(\sum_{i\in\omega} M_i)\ [S_0,\vee u_i,0]}$$

If $[S_0,b_0]M[S_1,b_1,u_0]$, then we say that $b_1$ is the *result* of the *evaluation* of M given $S_0$ and $b_0$. If $[S_0,0]M[\_,b_1,\_]$ then we write $M(S_0)=b_1$. (Underscores in expressions are variables whose possible values are obvious from the context, but which are omitted for clarity). Informally, an evaluation is the set of states and instructions that the machine will go through when started with a given set of data; the result is the signal it sends on to its subsequent component.

**Note that the execution of all MH machines is completed after a finite time. While Turing machines have the possibility of getting stuck in an infinite loop and never terminating, the same is not true of MH machines[10].**

**We now illustrate the notation by specifying and proving correct two elementary machines that have been discussed elsewhere [Hog96]. To 'prove a machine correct' is to prove that it behaves as expected. The first machine, $SAD_1$, can decide the truth of singly quantified arithmetical propositions such as $\exists x.P(x)$ for any primitive recursive predicate P that it is given. It does this by first testing P(1), then P(2), and so on. If any P($i$) happens to be true, then it sends a signal to the solution event. The second machine, $SAD_2$, can decide the truth of doubly quantified arithmetical propositions such as $\neg\forall x.\exists y.P(x,y)$. It is made up of an infinite string of $SAD_1$ components where the first component decides $\exists y.P(1,y)$, the second decides $\exists y.P(2,y)$, and so on.**

*Definition 1.6*. **Machines $SAD_1$ and $SAD_2$.**

**(i)** *$SAD_1$*. **We take P to be any primitive recursive predicate with one argument. Let $\mathsf{P}$ be the set of such predicates. The set $\mathsf{P}$ is countable [Göd31, footnote 7]. Define $S_{SAD1}=\mathsf{P}\times\mathsf{N}$. Let $I_{SAD1}$ contain just the single instruction NEXTX such that NEXTX((P,$x$),_)=((P,$x^+$),_,P($x$)). Define $SAD_1 \equiv MH(\sum NEXTX)$. Note that the instruction NEXTX is reasonable (definition 1.1) because, since P is primitive recursive, there is a Turing machine which can calculate P($i$) for any $i$ in a finite number of steps.**

**(ii)** *$SAD_2$*. **We take P to be any primitive recursive predicate with two arguments. Let $\mathsf{P}$ be the set of such predicates. Define $S_{SAD2}=\mathsf{P}\times\mathsf{N}\times\mathsf{N}$. Let $I_{SAD2}$ contain instructions NEXTX and NEXTY such that NEXTX((P,$x$,_),$b$)=((P,$x^+$,0),_,$\neg b$) and NEXTY((P,$x$,$y$),_)=((P,$x$,$y^+$),_,P($x$,$y$)). The machine is defined $SAD_2 \equiv MH(\sum$ {MH($\sum$NEXTY);NEXTX} ). Again, the instructions are reasonable.**

*Theorem 1.7*. **Correctness of $SAD_1$. For all primitive recursive propositions P with a single argument, [(P,0),_] $SAD_1$ [_,$\exists x.P(x)$,_]. That is, $SAD_1$ works as intended and can indeed decide the truth of any singly quantified arithmetical propositions.**

*Proof*. **By the definition of the instruction, NEXTX($i$,_) = ($i^+$,_,P($i$)). By STEP-TRAN, [$i$,_] NEXTX [$i^+$,_,P($i$)]. By MH-TRAN, [0,_] MH($\sum$NEXTX) [_,$\vee$ P($x$),_]. And because $\exists x.P(x) \Leftrightarrow \vee_x P(x)$, we deduce that [0,_] MH($\sum$NEXTX) [_,$\exists x.P(x)$,_]. So, for all propositions P, the result of the evaluation is $\exists x.P(x)$.** □

*Theorem 1.8*. **Correctness of $SAD_2$. For all primitive recursive propositions P with two arguments, [(P,0,0),_] $SAD_2$ [_,$\neg\forall$x.$\exists$y.P(x,y),_]. That is, $SAD_2$ can decide the truth of any doubly quantified arithmetical propositions.**

*Proof*. **First, by the definition of the instruction, NEXTY((($x$,$y$),_)=(($x$,$y^+$),_,P($x$,$y$)). By S-STEP and S-MH, [($x$,0)]MH($\sum$NEXTY)[($x$,0),$\vee_i$ P($x$,$i$),_]. Second, by the definition of NEXTX and by S-STEP, [($x$,0),$\exists y.P(x,y)$]NEXTX[($x^+$,0),_,$\neg\exists y.P(x,y)$]. Using S-SEQ to sequence the two parts, [($i$,0),0] MH($\sum$NEXTY);NEXTX [($i^+$,0),_,$\neg\exists y.P(i,y)$]. By S-MH, and performing the same final step as in the previous proof, we get [(0,0),_] MH($\sum${MH($\sum$NEXTY);NEXTX}) [(0,0),$\neg\forall x.\exists y.P(x,y)$,_]. So, for all propositions P, the result of the evaluation is $\neg\forall x.\exists y.P(x,y)$.** □

**[10] Since MH machines never fail to terminate, the transition rules describing the execution of the machines are similar in spirit as well as notation to Floyd-Hoare specifications for total**

## Machines which send only one signal.

**We have described machines where the signal arriving at any solution-event is deemed to be $\bigvee_i u_i$, the infinite disjunction of all signals sent to the solution event by all components. However, as mentioned in definition 1.2(ii) above, actual physical MH machines are allowed to receive at most one signal at their solution events: no 'swamping'. The following lemmas and theorems prove that this is an unimportant difference. Essentially, given some machine M, we can construct a new machine M' which keeps a flag to track whether a signal has so far already been sent, and only sends a signal if one has not already been sent. This involves expanding the set of states S to include the flag, adding a single new instruction RESET to reset it, and describing how M' is recursively constructed from M. The definitions and lemmas are technical and may be skipped: the important result is theorem 1.14, which states that machine M' does indeed behave as expected, and that the difference between M and M' is unimportant.**

*Definition 1.9.* **Suppose we have a machine M, with set of states S and set of instructions I. Create a new machine M' with set of states S' and set of instructions I' as follows.**

**Set of states: $S'=S\times B$.**

**Set of instructions: If $C\in I$ and $C(S_0,b_0)=(S_1,b_1,u_0)$ then $C'\in I'$ where $C'((S_0,h_0),b_0)=(S_1,h_0\vee u_0),b_1,u_0\wedge\neg h_0)$. Also, $RESET\in I'$ where $RESET((S_0,h_0),b_0)=((S_0,0),b_0)$**

**Recursive construction of new machine M' from M:**
**(STEP-MAP)** $C \rightarrow C'$
**(SEQ-MAP)** $M_0;M_1 \rightarrow M_0';M_1'$
**(MH-MAP)** $MH(\sum M_i) \rightarrow MH(\ RESET;\sum M_i'\ )$
**(recall that the $MH(C;\sum M_i)$ is shorthand for $MH(D_i)$ where $D_0=C$ and $D_{i+1}=M_i$).**

*Lemma 1.10.* **$\forall$M. If $[(S_0,h_0),b_0]$ M' $[(S_1,h_x),b_1,u_x]$ then $h_x=h_0\vee u_x$; and $\neg(h_0\wedge u_x)$**

*Proof*. **By induction on the structure of M.**
**Case STEP:** M was a single instruction C, derived from $C(S_0,b_0)=(S_1,b_1,u_0)$, so $[(S_0,h_0),b_0]$ C' $[(S_1,h_0\vee u_0),b_1,u_0\wedge\neg h_0]$, which satisfies the property.
**Case SEQ:** M is a sequence $M_0;M_1$. By the induction hypothesis we have $[(S_0,h_0),b_0]$ $M_0'$ $[(S_1,h_1),b_1,u_0]$ with $h_1=h_0\vee u_0$ and $h_0\Rightarrow\neg u_0$, and $[(S_1,h_1),b_1]$ $M_1'$ $[(S_2,h_2),b_2,u_1]$ with $h_2=h_1\vee u_1$ and $h_1\Rightarrow\neg u_1$. Sequencing them we get $[(S_0,h_0),b_0]$ $M_0';M_1'$ $[(S_2,h_2),b_2,u_0\vee u_1]$ which satisfies the property.
**Case MH:** M is in the form $MH(\sum M_i)$, so M' is $MH(\ RESET;\sum M_i'\ )$. By the rule MH-TRANS, $[\_,\_]$ M' $[\_,\_,0]$, which satisfies the property. □

*Lemma 1.11.* **Suppose we have $M'=MH(\sum D_i)$ with $[(S_0,h_0),b_0]$ M' $[(S_0,h_0),b_1,u_1]$ derived via rule MH-TRANS from $\forall i\in\omega$. $[S_i,b_i]$ $D_i$ $[S_{i+1},b_{i+1},u_i]$. We define $tot(x)=\sum_{i<x}u_x$. Then, $\forall x>1$. $tot(x)=h_x$. Essentially, $tot(x)$ is the total number of signals that have been sent up to that point.**

*Proof*. **M' must have been derived from rule MH-MAP: therefore $D_0$ is RESET, and $h_1=0$ and $u_0=0$; and every other $D_i$ is $M_{i+1}'$. By lemma 3.2, $h_{x+1}=h_x\vee u_x$. Let $mu_x$ be the signal $u$ that the original machine $M_{i+1}$ would have sent; again by lemma 3.2, $u_x=mu_x\wedge\neg h_x$. Therefore we have $tot(1)=0$ and $tot(x+1)=tot(x)+mu_x\wedge\neg h_x$. The proof that $\forall x>1$. $tot(x)=h(x)$ follows trivially by induction on x.** □

---

**correctness [Flo67].**

*Corollary 1.12.* **No machine M' ever has more than one signal arriving at a solution event.**

*Proof*. **The total number of signals tot($x$) that have been sent so far is always equal to $h_x$; and $h_x$ is a boolean value which can be only be 0 or 1.** □

*Lemma 1.13.* **If $[S_0,b_0]$ M $[S_1,b_1,u_1]$ then $[(S_0,h_0),b_0]$ M' $[(S_1,u_0 \vee h_0),b_1,u_0 \wedge \neg h_0]$.**

*Proof*. **By induction over the structure of M. The proof is similar to that of lemma 3.2 above, except for the case MH: Suppose that $M=MH(\sum M_i)$ with $[S_i,b_i]M_i[S_{i+1},b_{i+1},u_i]$ and, by the induction hypothesis, $[(S_i,h_i),b_i]$ $M_i$' $[(S_{i+1},h_i \vee u_i),b_i,u_i \wedge \neg h_i]$. Then M' is $MH(\sum D_i)$ with $D_0=$RESET and $D_{i+1}=M_i$. If $mu_x$ are the signals sent by the original $M_i$ components and tot($x$) is the total number of signals sent by machines $D_i$ so far, we prove by an induction similar to lemma 3.3 that $\forall x. \vee_{i<x} mu_x=$tot($x$), and so the signal $b_1$ sent on by machine M' is equal to the signal $b_1$ sent on by machine M. The other parts of the lemma follow directly from lemma 3.2.** □

*Theorem 1.14.* **Every machine M that takes the infinite disjunction of up to ω many signals sent to a solution event is computationally equivalent to some machine M' which never sends more than one signal to a solution event; and *vice versa*.**

*Forwards proof*. **By lemma 3.4, M' never sends more than one signal. By lemma 3.5, M' is computationally equivalent to M.**
*Reverse proof*. **M' is already a machine which takes the disjunction of up to ω many signals (although it happens that no more than one signal ever gets sent).** □

**With theorem 1.14 the final part of the argument has been provided, that the notation in this chapter is equivalent to the description of MH machines given by Hogarth [Hog96] and Earman and Norton [Ear94]. This concludes the setting out of the notation.**

## Summary

**The first part of this chapter set out a framework in which computations are viewed as a collection of instructions to be executed by a machine with state (definition 1.1). The framework is a general one which allows for different formalisms of computation to be expressed straightforwardly. It permits the straightforward expression of the Church-Turing thesis: that one particular formalism—namely that of Turing machines—can be taken as the standard measure of computation within a finite number of instructions.**

**The second part of this chapter extended consideration to non-finite computation with the aim of arriving at a formal logical notation. The justification for this project is that important results about the power of non-finite computation, in the next chapter, become evident only through careful structured analysis achieved with the notation. The first step was to exhibit a diagrammatic intermediate representation of MH machines. First the link between previous work [Hog96] and the diagrams was argued, then the link between the diagrams and the logical notation.**

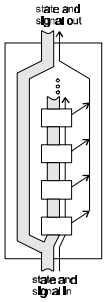


**Figure 9. How the signals connect.**

**The diagrams also make explicit the flow of information between components. Four restrictions on the flow of information that have been given in the literature were listed (definition 1.2) and shown to be met by the diagrams: lower boxes send signals to higher boxes or to solution events, downward signals are forbidden etc. One final restriction is that the solution event receive at most one signal—no swamping. This restriction is not satisfied by the diagrams. However it was later proved using the logical notation that the difference is unimportant: machines which can receive at most one signal are computationally equivalent to machines which can receive any number of signals (theorem 1.14). These results indicate that the diagrammatic notation given in this chapter does indeed correspond to the structure of MH machines given by [,].**

**The rest of the chapter presented the formal logical notation. Definition 1.2 gives the notation for the construction of the machines, where each machine is either a single instruction, a sequence of two other machines in order, or an infinite string of machines leading to a solution event. Definition 1.3 defines the denotation of a machine. This is the set of instructions, ordered in time, which the machine will execute. Chapter three goes on to prove that denotation of MH machines has countable ordinal type. Hogarth's axiom schema [Hog96, fig. 45] was reviewed and shown to correspond to the standard laws of ordinal multiplication and exponentiation.**

**Finally, definition 1.5 gives rules for the behaviour of MH machines upon execution. These rules are essentially a formal version of the diagrams. Two particular machines from the literature were expressed in the notation and proved correct—that is, it was proved that they behave as intended. The power of the notation is demonstrated by the fact that the machines can be completely and unambiguously described in a single concise expression, and by the fact that formal program verification is possible.**

**The result of this chapter is a three part logical notation for MH machines which has been shown to correspond to the structure of the machines given in the literature. The three-part notation is comprised of rules for the recursive construction of machines, rules for their denotation and rules for their behavior. The notation is used in the next chapter to arrive at results about the computational power of MH machines.**

# CHAPTER 2. COMPUTATIONAL POWER OF MH MACHINES

**The $SAD_1$ and $SAD_2$ machines, reviewed in the previous chapter, are capable of deciding the truth of arbitrary arithmetical sentences in the form $\exists x.P(x)$ and $\forall x.\exists y.P(x,y)$ respectively. The $SAD_1$ machine works by testing every value of $x$ in order and sending a signal when a counter-example is found; the $SAD_2$ machine tests every value of (x,y) in lexicographical order—that is, in the order (0,0), (0,1), (0,2), …, (1,0), (1,1), …, …. It has been proved that $SAD_n$ machines form a strict hierarchy: $SAD_1$ can decide sentences with a single quantifier but no more; $SAD_2$ can decide sentences with two quantifiers but no more; and so on [Hog96, prop.3.12.2]. And the AD machine, which has an infinite string with $SAD_1$, $SAD_2$, …, can decide all arithmetical sentences.**

**However the proof only applies to the $SAD_n$ machines as specifically defined in Definition 1.6 and does not apply to MH machines in general. Indeed, there exists a machine that is the same size as a $SAD_1$, but is as powerful as an AD machine in that it can decide all arithmetical sentences. We call this machine the *Arithmetic-Powerful-$SAD_1$*, or $APS_1$. The key difference is as follows. The $SAD_1$ machine, when given a proposition P, goes through all values of $x$ in order and sent a signal when it finds an $x$ for which P($x$) is true. But the $APS_1$ machine contains a table of all possible propositions $Q_i$ (including all quantified propositions), along with an indication of whether each is true or false; when given a quantified proposition Q, it simply looks it up in its table. To describe the machine concisely, we use the fact that all arithmetical propositions Q have some Gödel number #Q which uniquely identifies the proposition**
**p.7].**

> *Definition 2.1.* **$APS_1$: Let the set of states $S_{APS1}=N$. We write $i_Q$ for the *indicator function* of the (possibly quantified) proposition Q: $i_Q$ is 0 if the proposition is false, and 1 if the proposition is true. Define the set of instructions $I_{APS1} = \{ C_{\#Q}: C_{\#Q}(\#Q,\_)=(\#Q+1,\_,i_Q) \}$. Clearly every instruction $C_x$ is reasonable (def. 2.1). The machine is defined $APS_1 \equiv MH(\sum_{x\in\omega}C_x)$.**
>
> *Theorem 2.2.* **Correctness of $APS_1$: $[\#Q,\_]\ APS_1\ [\_,i_Q,\_]$. That is, the machine $APS_1$ can decide the truth of all arithmetical propositions.**
>
> *Proof*. **Straightforward.** □

**The $APS_1$ machine might be considered 'unfair' for two reasons. The first reason is that it is an ω-machine which has countably many different instructions, and so it cannot be equivalent to a single Turing machine heading towards a solution event. The proof of this is that such a Turing machine would be able to solve the halting problem simply with table indicating whether each numbered Turing machine halts[11]. However, as discussed in the previous chapter, this is an unimportant restriction that only has any effect at the lowest level of machine. It is easy to construct another machine $APS_2$ of size $\omega^2$, essentially the same as $APS_1$ but with each instruction replaced by an ω-component, which calculates exactly the same result. In $APS_2$ there does not exist any ω-component not equivalent to a Turing machine, and there are only finitely many instructions. Since there are no restrictions yet on the sequence in which ω-components can be composed in an infinite string, the first reason for calling the $APS_1$ machine unfair has been circumvented.**

---

**[11] For a machine H to solve the halting problem means that, given the number #A corresponding to some other Turing machine A and some data D, the computation H(#A,D) always terminates and gives the result 0 or 1 according to whether or not the computation A(D) would have terminated.**

**The second reason for calling the machine unfair arises from the vague suspicion that the machine is somehow too complicated: that it would have an infinitely large description. However we must be a little more precise in framing this objection, since the machine above was described in only four lines of text—manifestly a finite description!. The objection is that it would not be possible to use any regular system of finite description—a programming language—composed only of basic elementary descriptive units, and still have a finite description of the $APS_1$ machine. This finite description can be taken without loss of generality to be a natural number. Effectively, we wish to find a way to assign Gödel numbers to MH machines[12].**

## Gödel numberings for MH machines

*Definition 2.3*. **Gödel numberings.**

**(i)** **A Gödel numbering # is an injection from machines** M **to natural numbers** N**. That is, it is a way of assigning a unique number to different machines.**

**(ii)** **We write dom# for the domain of the numbering. This is the set of machines that are assigned numbers.**

**(iii)** **The terms** *numbering*, *system of finite description* **and** *programming language* **are equivalent.**

**Finding a Gödel numbering for the set of Turing machines is straightforward. This is because there are countably many Turing machines. This is clear because the numbering exhibited by Turing [sec5] is capable of describing all Turing machines. However, as shown by the following lemma, the situation is more complex with MH machines: there is an uncountable number of them.**

*Lemma 2.4.* **There are uncountably many different machines in the set** M **of MH machines.**

*Proof*. **Suppose there is an injection # from all machines in** M **onto different** N**. A machine $MH(\sum M_i)$ is characterised by the countably infinite sequence of numbers $\#(M_i)$. And the set of infinite sequences of numbers is uncountable, by a diagonal argument. Therefore there are uncountably many different $MH(\sum M_i)$, and so dom# cannot be countable. Contradiction.** □

**Note that the diagonal argument still applies even if we allow only a finite instruction set.**

**The fact that there are uncountably many different machines means that no numbering # can cover every single machine: we will have to restrict our attention to only a countable subset of machines. But the question remains as to precisely which countable subset of** M **to consider. Suppose we start with only two different machines named '0' and '1'. Consider the different possible sequences of components $M_i$ in $MH(\sum M_i)$ where each component is either '0' or '1'. Lemma 2.5 below implies that there is no maximal countable set of such sequences: any**

---

[12] **The purpose of a Gödel numbering is to express machines in such a way that they can be operated upon by other machines as data. In his paper on the Incompleteness Theorem [Göd31], Gödel was interested in arithmetical propositions about numbers: so, he found a way to express arithmetical propositions themselves as numbers. The MH machines in which we are interested operate upon a set of states** Q**: so, we wish to express each MH machine as an element of** Q**. Without loss of generality, because** Q **is countable (def. 2.2), we will take** Q **to be the set of natural numbers** N**.**

**countable subset that is proposed could be improved upon by expanding it to describe even more machines. Therefore the question of which single countable subset of M should be considered has no satisfactory answer. This makes the choice of a numbering for MH machines problematic.**

> *Lemma 2.5.* **The partially ordered set of countable sets of sequences has no maximum element.**
>
> *Proof*. **Consider the real number r, where $0 \le r \le 1$. Then r has an infinite binary expansion (r) such as, for instance, 0.010110101… Let the set of sequences $A_r$ contain only the single element (r). $A_r$ is clearly countable (having only one element) and is a subset of the set of sequences of numbers; and $A_r$ are disjoint for every r. Suppose there exists a maximal element A of the partially ordered set of countable sequences. Since there are uncountably many $A_r$, not all can be contained in A. Let $A_x$ be one such. $A \cup A_x$ is countable and is larger than A: therefore A could not have been a maximal element, and neither could it have a maximum.** □

**Since there is no single countable subset of M (that is, no dom#) that is greater than all the others, it is desirable to prove theorems which hold for a range of different numberings #. Some numberings are trivial and useless (such as the null numbering, in which no machine is assigned a number). In this dissertation only certain classes of numbering will be considered. The three classes, explained in definition 2.6 below, will be called** *sequencing-complete*, *simulation-complete* **and** *Turing-generable*.

## Solutions to the problem of uncountably many Gödel numberings

**Before continuing with the investigation of these arbitrary restrictions[13], it is useful to review the reasons for their introduction. The problem raised at the start of this chapter is that some very powerful machines are felt, intuitively, to be too complicated. It might be possible for the machines simply to exist, laid out in space-time with the appropriate infinite program already in place, but it is felt that it would not be possible to write a finite description of them within a systematic programming language. It is therefore necessary to consider only a countable subset of machines. Lemma 2.5 shows that there is no single best numbering, no single best countable subset. Therefore it is desirable that proofs be valid for a range of different numberings. A numbering of machines would not be useful if it were not possible to describe every single instruction $C \in I$; it would also not be useful if we could describe $M_0$ and $M_1$ but could not also describe $M_0;M_1$. A numbering in which it is possible to do both is called a** *sequencing-complete* **numbering.**

**One way of considering a system of finite description, or a numbering, is that it makes it possible for some machine S, given the number of another machine M, to simulate its operation and to give whatever output M would have given: such numberings will be termed** *simulation-complete*. **Turing machines satisfy this property: the universal Turing machine, when given a description of any other Turing machine, can simulate it [Tur37, section 7]. Gödel used the fact that his numbering of arithmetic was simulation-complete to prove the Incompleteness Theorem [Göd31, theorem IX]. But it is not clear how to apply the result to MH machines. It might be that, given any machine M of size $\alpha$, we require that a simulating-machine of the same size can simulate it; or we might say that the simulating machine is smaller or larger. The theorems in this dissertation concern the case, chosen arbitrarily, where the**

---

[13] **The use here of the term 'arbitrary' in relation to numberings of ordinal computation, is unrelated to Turing's use of the term in his paper on ordinal logics [Tur39, section 11].**

**simulating-machine is the same size. This means that the simulation requires roughly the same computational power to simulate a machine, as that machine would itself have required.**

**A third way of describing MH machines has also been proposed whereby each infinite string of components can be characterised by a single Turing machine T [Wis96]. This Turing machine computes the numbers for each component. So, T(0) is the number for the first component; T(1) is the number for the second component; and so on. This will be termed a** *Turing-generable* **numbering. It is certainly the case that a Turing-generable numbering describes the construction of an MH machine in terms of the basic building blocks of Turing computation. However, there seems to be no reason to suppose that these numberings are the only way—or even the most general way—of specifying MH machines. For instance, we might instead consider** *ω-generable* **numberings in which the number of each component can be generated by an ω-machine.**

**The restriction to** *sequencing-complete* **numberings is justified because any numbering that was not sequencing-complete would not be useful. The restriction to** *simulation-complete* **numberings is more arbitrary, but is worth investigating because it leads to interesting results and because it relates to Gödel's initial intent in introducing numberings. The restriction to** *Turing-generable* **numberings is the most arbitrary: it is considered in this dissertation because it is only with this restriction that there exists a general correspondence between MH machines and arithmetic.**

**The complexity of numbering MH machines sheds light on the issue of numbering conventional Turing machines. For instance, it is common in the literature for some Turing-machine numbering to be presented as 'obvious', with the existence of a universal Turing machine that can simulate all others being derived as a consequence. But, as the general MH case illustrates, it would perhaps make more sense to think of the existence of a universal Turing machine as being the basic requirement, and the numbering scheme as being derived as a consequence.**

*Definition 2.6.* **Different families of numberings.**

**(SEQ-COM)** **A numbering # is** *sequencing-complete* **when, if $C \in \mathbb{I}$, then $C \in \mathrm{dom}\#$; and when, if $M_0, M_1 \in \mathrm{dom}\#$, then $M_0;M_1 \in \mathrm{dom}\#$; and** *vice versa***. Essentially, every instruction on its own is part of the numbering; and if two machines are numbered then the machine obtained through their sequential composition is also numbered. This numbering makes no restriction on infinite strings of machines.**

**(SIM-COM)** **A numbering # that is** *simulation-complete* **is one that is sequencing-complete and in which, for every limit ordinal $\lambda$, there exists a limit machine $S \in \mathrm{dom}\#$ of size $\lambda$ such that for every machine $A \in \mathrm{dom}\#$ that is not larger, $S(\#A,D)=A(D)$. Essentially, S is a simulating machine that can simulate the operation of any machine that is smaller or the same size.**

**(TG-COM)** **A numbering # that is** *Turing-generable* **is one that is sequencing-complete and in which, for every Turing machine T that takes a single number** *i* **as input, $(\forall i.\ T(i)=\#M_i) \Rightarrow MH(\sum_i M_i) \in \mathrm{dom}\#$. Essentially, if we can have a Turing machine that generates the number for every single component in a string of components, then that string of components is also numbered. We further require that, given any machine #M, it is possible for a Turing machine to calculate its deduce its constitutent components, and that there is a Turing machine that can simulate any instruction C given its number.**

**The following sections each assume different families of numbering and deduce consequences of that numbering. First we consider numberings in general, then those that are sequencing-complete numberings, then those that are simulation-complete, and finally those numberings**

**that are Turing-generable. The results are then assembled to show that the successive restrictions on numberings form a strict hierarchy. This gives rise to a non-finite addition to the Church-Turing thesis.**

## Consequences of the assumption of a numbering

**We first simply consider some numbering, without restriction to any of the above families. From this assumption it can be deduced that there does not exist a** *negating-machine* **N which can simulate all other machines[14] and invert the output: N(#M) = ¬M(#M). This sentence is essentially similar to standard liar paradox, "This sentence is not true", and represents a Gödel sentence for MH machines.**

*Definition 2.7.* **A negating machine N is one such that N(#M) = ¬M(#M).**

*Theorem 2.8.* **No negating machine N exists such that N∈dom# and N can negate all numbered machines (for any numbering).**

*Proof*. **N(#N) is contradictory.** □

## Consequences of the assumption of a sequencing-complete numbering

**The previous result was proved without any assumption about the type of numbering. We now make the additional assumption that the numbering in question is sequencing-complete: if any $M_0$ and $M_1$ are numbered, then so is $M_0;M_1$. From this assumption we deduce that there does not exist a universal MH machine. (Just as a universal Turing machine is able to simulate the operation of any numbered Turing machine it is given, so too a universal MH machine would be able to simulate the operation of any given MH machine). The proof uses the fact that the existence of a universal MH machine would permit the construction of a universal negating machine.**

**The result points to a subtle difference between Turing computation and MH computation. Turing machines need not always halt, and there does exist a universal Turing machine. MH machines, on the other hand, always arrive at an answer; and it is because of this that there does not exist a universal MH machine.**

*Corollary 2.9.* **No universal MH machine S exists such that S∈dom#, and S(#A,D)=A(D) (assuming a sequencing-complete numbering).**

*Proof*. **Suppose that such a machine S does exist. Consider the machine D;S;N where D(x,_)=((x,x),_,_) and N(_,*b*)=(_,¬*b*,_). By SEQ-COM, D;S;N∈dom#. Now, D;S;N(#D;S;N) = (by SEQ-TRAN) ¬S(#D;S;N,#D;S;N) = (by defn. of S) ¬D;S;N(#D;S;N). Contradiction; therefore no such S can exist.** □

---

[14] **Note that the requirement is that the universal negating machine should be able, given the number of any other machine, to negate it. This is much stronger than the requirement that, for any machine M, there should exist some other machine $N_M$ which can negate M. After all, M(#M) must evaluate either to 0 or to 1; if it evaluates to 0, then the simple machine which outputs a 1 has negated it; if it evaluates to 1, then the simple machine which outputs 0 suffices. The argument given in [Wis96] for an infinite hierarchy of non-finite computation is flawed because it establishes only the second form of the requirement rather than the first.**

## Consequences of the assumption of a simulation-complete numbering

**We now make the additional assumption that the numbering in question is simulation-complete. This permits a proof of the *size-power theorem*, relating the size of an MH machine to its computational power: the larger the machine, the more powerful it is. It should be noted that the size-power theorem does not hold unless we make the assumption of simulation-completeness. The significance of this result is discussed at the end of the chapter. Before giving the proof, it is first necessary to define equality of computational power between machines.**

*Definition 2.10.* **Equality of machines. $M_0=M_1$ means that $\forall S_0,b_0.\ [S_0,b_0]\ M_0\ [S_1,b_1,u_0] \Leftrightarrow [S_0,b_0]\ M_1\ [S_1,b_1,u_1]$. Essentially, if two machines are computationally equal, then they produce exactly the same answers as each other for all possible input values.**

*Theorem 2.11.* **Size Power Theorem**

**(POWER)** $\forall\alpha.\ \forall M_0.\ ord[[M_0]]=\alpha \Rightarrow \exists M_1.\ ord[[M_1]]>\alpha \wedge M_1=M_0$.
**For every machine of size $\alpha$, there exists a machine of greater size that is as powerful.**

**(STRICT)** $\forall\lambda.\ \exists M_0.\ ord[[M_0]]=\lambda \wedge \neg\exists M_1.\ ord[[M_1]]<\lambda \wedge M_0=M_1$. ($\lambda$ is a limit ordinal)
**For every limit ordinal $\lambda$, there is some machine $M_0$ of that size such that no smaller sized machine is as powerful. Effectively, machine $M_0$ takes full advantage of its size.**

*Proof of power*. **Given some machine $M_0$, construct $M_1 \equiv M_0;\text{NULL}$ where $\text{NULL}(S,b)=(S,b,)$. By SEQ-COMP, $ord[[M_1]]=ord[[M_0]]+1$, which is greater than $ord[[M_0]]$. By SEQ-TRAN, $M_0=M_0;\text{NULL}$. By SEQ-COM, $M_0;\text{NULL} \in dom\#$. We have thus exhibited a machine $M_1$ that satisfies the conditions.**

*Proof of strict*. **Define $M_0=D;S_\lambda$, where $D(x,\_)=((x,x),\_,\_)$, $ord[[S_\lambda]]=\lambda$, and $S_\lambda(\#A,D)=A(D)$ for all $ord[[A]]\leq\lambda$. By SIM-COM, such a machine $S_\lambda$ exists and is an element of dom#; by SEQ-COM, $D;S_\lambda \in dom\#$. By SEQ-COMP, $ord[[M_0]]=1+\lambda$; since $\lambda$ is a limit ordinal, $ord[[M_0]]=\lambda$. Suppose there exists a machine $M_1$ such that $ord[[M_1]]=\alpha$, $\alpha<\lambda$, and $M_0=M_1$. Therefore $M_1(\#A)=A(\#A)$ for all $ord[[A]]\leq\lambda$. Consider $M_1;N$ where N is as defined in corollary 2.9. By SEQ-COMP, $ord[[M_1;N]]=\alpha+1$; since $\alpha<\lambda$ and $\lambda$ is a limit ordinal, $\alpha+1<\lambda$. By SEQ-COM, $M_1;N \in dom\#$. From definition of N and by SEQ-TRAN, $M_1;N(\#A)=\neg A(\#A)$ for all $ord[[A]]\leq\lambda$. Therefore $M_1;N(\#M_1;N) = \neg M_1;N(\#M_1;N)$. Contradiction: therefore no such $M_1$ exists.** □

## Consequences of the assumption of a Turing-generable numbering

**We finally assume that the numbering in question is Turing-generable. That is, we assume that the infinite strings of components may be characterised by a Turing machine T where T(0) generates the number for the first component, T(1) generates the number of the second component, and so on. The difficulty encountered above (lemma 2.5), of having no maximal numbering, no longer applies (lemma 2.12). This means that we need not prove results over all numberings: it is sufficient merely to prove results about any particular maximal numbering. In**

this respect Turing-generable numberings behave in the same way as numberings that are used for conventional Turing machines.

*Lemma 2.12.* **The numbering in which the infinite string MH($\sum_i M_i$) has the Gödel number #T, with T a Turing machine satisfying T($i$)=#$M_i$, is maximal over all Turing-generable numberings. That is, there is no other Turing-generable numbering which contains more machines.**

*Proof.* **From definition 2.6, every Turing-generable machine is an element of the numbering.** □

*Lemma 2.13.* **All Turing-generable machines are simulation-complete.**

*Proof.* **Given a universal Turing machine, it is possible (given the numbering in theorem 2.12) to construct a simulating machine S such that S(#A,D)=A(D) for all numbered machines A of equal or smaller size. Simulating a single instruction is straightforward. When simulating an infinite string of components, the machine S simply evaluates T($i$) for each component $i$ and recursively simulates that component. The proof that it can indeed simulate all machines of equal or smaller size is by induction over the structure of #A.** □

**The important result in this section is that by restricting attention to Turing-generable numberings, it becomes possible to demonstrate an equivalence between the size of machines and the Kleene arithmetical hierarchy. In particular, a machine of size $\omega$ can decide all arithmetical sentences with a single quantifier $\forall x.$P($x$), where P can be any partial recursive predicate that terminates for all values of $x$, but no smaller machine can decide $\forall x.$P($x$). A machine of size $\omega^2$ can decide all arithmetical sentences with two quantifiers, but no smaller machine can. And likewise for higher exponential powers.**

**The proof is similar to that due to Hogarth [Hog96, lemma 3.12.5]. He exhibited a family of machines $SAD_n$ of size $\omega^n$ in which arithmetical sentences with $n$ quantifiers can be decided by the $SAD_n$ machine but by nothing smaller. However, his result applied only to the particular machines $SAD_n$ that he had defined[15]. The result in this section is a more general result: not only is it impossible for SAD machines smaller than $\omega^n$ to decide sentences with $n$ quantifiers, but it is also impossible for *any* machine of size smaller than $\omega^n$ to decide such sentences.**

**When proving that some task is impossible for conventional Turing machines, the argument generally used is as follows. If such a task could be achieved by some Turing machine M, then it would be possible to construct from M some other Turing machine M' which could solve the halting problem. However, the there exists no Turing machine that can solve the halting problem. Therefore the task is impossible. This form of argument is termed 'reduction to the halting problem'. The MH version of the argument is 'reduction to the negating machine', using the fact (theorem 2.8) that an MH negating machine does not exist. We use this technique to prove the following theorem.**

*Theorem 2.14.* **Arithmetic hierarchy theorem.**
**No machine smaller than $\omega^n$ can decide an arithmetic sentence with $n$ quantifiers in the form $\forall x.\exists y\ldots$ P($x$,$y$,…) where P is a partial recursive function that always terminates.**

*Proof.* **By induction, using reduction to the negating machine to prove the base case and the induction step. The proof is lengthy and has the same overall structure as that of**

[15] **Hogarth's result also applied only to primitive recursive functions, rather than partial recursive functions that always terminate. The difference is not important.**

**Hogarth [3.12.5]. We therefore give only the first two steps, as an indication of how to proceed.**

*First case: machines up to size* ω. **Suppose that there exists a machine D(#P), smaller than ω, which can decide the truth of the arithmetical sentence $\exists x.P(x)$, with P any partial recursive predicate that always terminates. Suppose we have some machine M of size smaller than ω. We wish to construct from #M some arithmetical proposition $sig_M$ so that by solving $\exists x.sig_M$, the machine D will also be solving M(#M). By the definition of Turing generability, from the number #M it is possible to compute the Gödel number for each of the machine's instructions with a Turing machine and also, by implication, with some terminating partial recursive proposition. The fact that the computational formalism is reasonable (definition 1.1) implies that there is some universal Turing machine U, where U(#A,D)=A(D), that can simulate each instruction. Consider the evaluation of M(#M). The state after the first instruction is $U(\#I_0,\#M)$ where $\#I_0$ is the number of the first instruction; the state after the second is $U(\#I_1,U(\#I_0,\#M))$; etc. In this manner it is possible to construct a partial recursive expression $P_M$, which evaluates to the in which the machine will be after having executed all instructions in M. From $P_M$ it is possible to construct a partial recursive expression $sig_M$ which indicates whether machine M sends an output signal to a subsequent component. Hence, $D(\#sig_M)=M(\#M)$ for all machines M of size smaller than ω. From D we can construct a negating machine D'(#$sig_M$), still smaller than ω, which evaluates to ¬M(#M). This leads to the negating-machine contradiction. Therefore no such machine D can exist that can decide $\exists x.P(x)$ with fewer than ω instructions.**

*Second case: machines up to size* $\omega^2$. **Suppose there exists a machine D(#P), smaller than $\omega^2$, which can decide the truth of arithmetical sentences $\forall x.\exists y.P(x,y)$ where P can be any partial recursive predicate that always terminates. Suppose we have some machine M of size smaller than $\omega^2$ and (without loss of generality) larger than ω. This machine must consist of a string of components with at least one of the components an ω-component. We wish to construct the function sim($x$) which evaluates the state up to but not including component $x$. By the definition of Turing generability, it is possible given #M to compute the Gödel number $\#M_x$ for each component. If a component $M_x$ is a simple instruction, then we simulate it as per the previous case. If a component $M_x$ is an ω-component, then we can construct an expression of the form $\exists y.P(x,y)$ or $\forall y.P(x,y)$ for whether that ω-component sends a signal to the subsequent component $M_{x+1}$. Construction of the function sim($x$) is straightforward given the two cases, and the proof proceeds in the same way as that for machines up to size ω.** □

## The hierarchy of possible numberings

**In the previous sections we have reviewed different possible numberings of machines. First we considered numberings in general, in which it was proved that no negating machine exists. The additional assumption of a sequencing-complete numbering leads to the conclusion that no universal MH machine exists. The further assumption a simulation-complete numbering leads to the rank-power theorem, that the power of a machine increases with its size; and finally the assumption of a Turing-generable numbering leads to a relation between the size of MH machines and the Kleene arithmetical hierarchy. Theorem 2.15 proves that these different levels of assumptions give rise to a strict hierarchy, with Turing-generable numberings covering the smallest class of computation. The implications of this hierarchy are discussed below.**

*Theorem 2.15.* **Hierarchy of numberings.**
**To compare families of numberings A and B, we write A≤B if every computation that is in the domain of some numbering in A, has an equal computation (definition 2.10) in the domain of some numbering in B. We write A<B if A≤B, and there exists at least one computation in a numbering in B that has no equal in the domain any numbering in A.**

| | | | |
|---|---|---|---|
| | | **Turing-generable #** | **(arithmetical sentence deciding computations)** |
| **(i)** | **<** | **simulation-complete #** | **(size-power theorem)** |
| **(ii)** | **<** | **sequencing-complete #** | |
| **(iii)** | **<** | **numberings # in general.** | |

*Proof of (i).* **By lemma 2.13, the family of Turing-generable numberings is smaller than or equal to the family of simulation-complete numberings. To prove that the computation of Turing-generable numberings is strictly smaller, consider the ω-machine $MH(\sum C_i)$, in which $C_{\#M}(\#M,\_)=(\#M^+,\_,h_M)$ with $h_M$ equal to 1 if the Turing machine M halts, and 0 otherwise. This can be part of a simulation-complete numbering. However, if it were part of a Turing-generable numbering then the same Turing machine which generated the instructions would also be able to solve the halting problem.**

*Proof of (ii).* **Straightforward, since sequencing-complete numberings make no restriction on the composition of infinite strings of components whereas simulation-complete numberings do.**

*Proof of (ii).* **Sequencing-complete numberings do not have universal simulation machines (corollary 2.9). The same is not true of all numberings.** □

**The result of this chapter is a strict hierarchy of numberings with the following properties.**

- **(i) Arithmetical sentence-deciding computation comes at the level of Turing-generable numberings. (That is, every arithmetical-sentence deciding computation is equal to some computation in a Turing-generable numbering).**
- **(ii) The size-power theorem lies somewhere between simulation-complete and sequencing-complete numberings. (All simulation-complete numberings satisfy the size-power theorem; but there may be sequencing-complete numberings that are not simulation-complete that also satisfy the theorem).**

**The numberings as presented were progressively more arbitrary, with the restriction to Turing-generable numberings being the most arbitrary. The absence of a universal MH machine is the most secure result. The size-power theorem appears reasonably secure since its assumption of simulation-completeness reflects the Gödel's original intention in introducing numberings. But the relation between the size of MH machines and equivalent arithmetical sentences [Hog96, prop. 3.12.5] appears at present to be arbitrary and without justification.**

**Of the families of numberings introduced, it is only that of Turing-generable numberings that would reasonably be considered to describe the construction of machines in terms of basic descriptive units. However we can not rule out the possibility that there may be different families of numberings, equivalent perhaps to simulation-complete or sequencing-complete, that could also be considered reasonably to be composed of basic descriptive units. It is also possible that a justification can be found for restricting attention to Turing-generable numberings. This would reinstate the relation between size of MH machines and arithmetical sentences.**

**The question of which numbering to adopt out of those in the hierarchy is an important one, because computational power changes with position in the hierarchy. But nothing in this**

**chapter recommends any particular numbering over any other. This problem gives rise to a version of the Church-Turing thesis that has been extended to cover non-finite computation.**

*Thesis 2.16.* **Church-Turing thesis extended to non-finite computation.**

**(i) There are many different formalisms for computations of a finite number of instructions, but they are all equivalent (and in particular they are all equivalent to Turing machines) so it does not matter which is chosen.**

**(ii) There are many different numberings for computations of a non-finite number of instructions. Many of them have different computational power. The question of which to chose is therefore important; and there is no particular reason to chose any individual one over the others.**

**The extended Church-Turing thesis leads to import philosophical questions about the overall problem of non-finite computation. It might be possible for a machine to exist, already laid out in space-time, that could solve all problems; but it would not be possible systematically to describe this machine. It might be that we should term 'computable' those problems that can be solved by some machine that can exist; or it might be that we should term computable only those problems that could be solved in principle by some machine with a finite description; or it might be that we should term a problem computable only if the answer could be known within a finite time to some observer who also programmed the machine. It is also not clear how the machines should be programmed: if the physical situation is such that they can not be programmed, then the restriction to finite programs would appear to be irrelevant. If we allow for non-programmable machines, then there is no longer even a Gödel sentence for the system.**

**The starting point for this dissertation was the physical construction of MH space-times. In the two chapters above, a formal logical analysis has been presented of the machines that operate within these spacetimes. Now, as indicated in the previous paragraph, the problem has become a philosophical one. We leave it open to further discussion.**

# CHAPTER 3. UNDERLYING PRINCIPLES OF COMPUTATION

**In this chapter an underlying abstract principle of computation is proposed, on the basis of the preceding analysis of MH machines.**

**We will consider only** *sequential deterministic* **computation. 'Sequential' means that instructions can never happen in parallel. 'Deterministic' means that there will never be more than a single next instruction to chose from. A** *computation* **is a collection $C$ of instructions to be executed, totally ordered in time by $<$. (If $C_0$ is to be executed at any time before $C_1$, we write $C_0<C_1$).**

**The essential property of a computation is that, among the collection of instructions left to be executed, there will always be a single instruction to be performed first (unless the process has terminated and the collection of remaining instructions is empty). This simple restriction implies that all computations have ordinal type; equivalently, the collection of instructions in a computation is well-founded.**

> *Definition 1.1.* **An ordered set Q of instructions is a computation if and only if it is a totally ordered set and, at any stage, there is a unique next instruction to be performed.**
>
> *Theorem 3.2.* **A set Q of instructions is a computation if and only if it has ordinal type.**
>
> **To prove the theorem concisely we introduce an auxiliary definition: a predicate T is a** *yet-to-be-executed* **predicate if it divides an ordered collection of instructions into two contiguous parts, such that $T(x)$ is false for all instructions in the first part and true for all instructions in the second. That is $\forall a,b.\ \neg T(a)\wedge T(b) \Rightarrow a<b$. An alternate way to define a computation Q is with the property that for all yet-to-be-executed predicates T, the set $\{x\in Q: T(x)\}$ of instructions still to be executed has a minimal element.**
>
> *Forward implication (only if).* **By the definition of computation, the computation Q is a totally ordered set; it remains to be shown only that every subset A of Q has a minimal element. Suppose that A is a counter-example: a subset of Q without a minimal element. We define the predicate $T(x) \equiv \exists a\in A.\ a\le x$. The predicate T is a yet-to-be-executed predicate. (Proof: assume $\neg T(x)\wedge T(y)$. Deduce $\forall a\in A.a>x \wedge \exists b\in A.b\le y$. Putting $x=y$ or $x>y$ gives a contradiction; therefore $x<y$.) Because T is a before-after predicate, we deduce that $\{c\in Q: \exists a\in A.a\le c\}$ has a minimal element: call it $m_A$. Because $m_A$ is an element of that set it must satisfy $\exists a\in A.a\le m_A$; because it is a minimal element, it must satisfy $\forall a\in A.m_A\le a$; therefore $m_A\in A$. However, we assumed that the subset A has no minimal element: $\neg\exists a\in A.\forall b\in A.a\le b$. Contradiction: therefore no counter-example A exists.**
>
> *Reverse implication (if).* **We assume that Q has ordinal type. Therefore, every subset has a minimal element. In particular, for every yet-to-be-executed predicate T, the subset $\{c: T(c)\}$ has a minimal element.** □

**This result gives** *a priori* **grounds for believing that all possible computations are ordinals. There are two particular subsets of the class of computations that will be considered. The first subset contains all countable computations. It will be shown that all MH machines perform countable computations (theorem 3.6), and that there are good reasons for dismissing computations that are non-countable (theorem 3.7). The second subset is that containing only finite computations.**

*Definition 3.3*. **Types of computation.**

**(i)** **An** *ordinal computation* **is simply a computation, as defined above.**

**(ii)** **A** *countable computation* **is one which has no more elements than there are natural numbers.**

**(iii)** **A** *finite computation* **is one which has a finite number of elements.**

*Theorem 3.4.* **Writing** $FC$ **for the set of finite computations,** $CC$ **for the set of countable computations, and** $OC$ **for the class of ordinal computations,** $FC \subset CC \subset OC$**.**

*Proof*. **Hartog's lemma states that for any set, there exists an ordinal that cannot be mapped injectively onto it [Joh93, p. 78]. Therefore there exists an ordinal computation which cannot be mapped injectively onto the set of natural numbers.** □

*Lemma 3.5.* **In the set** $\mathsf{S}$ **with order relation** $\leq$ **generated by the following axioms, every element of** $\mathsf{S}$ **has countable ordinal type; and there is no countable ordinal not isomorphic to some element of** $\mathsf{S}$**. In other words,** $\mathsf{S}$ **is equivalent to the set of all countable ordinals. (Theorem 3.6 shows that MH machines have the same structure as** $\mathsf{S}$**. Properties that hold for** $\mathsf{S}$**, such as being countable, also hold for the set of MH machines).**

**(STEP-CC)** $\varnothing \in \mathsf{S}$; $\varnothing \leq \varnothing$

**(SEQ-CC)** $\alpha,\beta \in \mathsf{S} \Rightarrow \alpha+\beta \in \mathsf{S}$**, where** $\alpha+\beta$ **is the disjoint union** $\{0\}\times\alpha \cup \{1\}\times\beta$

**(MH-CC)** $\forall i.\ \alpha_i \in \mathsf{S} \Rightarrow \sum_{i\in\omega}\alpha_i \in \mathsf{S}$**, where** $\sum_{i\in\omega}\alpha_i$ **is the disjoint union** $\bigcup_i \{i\}\times\alpha_i$

**The disjoint unions are ordered lexicographically:** $(a,b)\leq(c,d) \Leftrightarrow a<b \vee (a=b \wedge c\leq d)$**.**

*Forward implication (every element of S is countable)*. **Note that the class** $ON$ **of all ordinals satisfies the three rules STEP-CC, SEQ-CC and MH-CC; since the class** $\mathsf{S}$ **was defined to be the smallest collection satisfying the three rules,** $\mathsf{S}\subseteq ON$**. It is a standard result that all sub-collections of** $ON$ **are well-founded. Therefore,** $\mathsf{S}$ **is well founded and we can use the principle of transfinite induction in proofs about** $\mathsf{S}$**. However, this result about** $\mathsf{S}$ **being well-founded is incidental and not needed by the proof, since we only need a straightforward rule induction. STEP-CC is straightforward, as is SEQ-CC. It remains to prove the case for MH-CC: that if** $\alpha_i$ **are countable for all** $i\in\mathsf{N}$**, then so is** $\sum_i\alpha_i$**. The fact that each** $\alpha_i$ **is countable means that there for each** $i$ **there exists a function** $f_i$ **which maps all elements** $\alpha_i$ **onto** $\mathsf{N}$**. Given the axiom of choice, we construct a function** $g(i,\alpha)=2^i.3^{f_i(\alpha)}$**, mapping all elements of the disjoint product** $\sum_i\alpha_i$ **onto the natural numbers. This function g can be seen to be injective: therefore** $\sum_i\alpha_i$ **is countable. By the principle of rule induction, all elements of the set** $\mathsf{S}$ **generated by the three rules are countable.**

*Reverse implication (every countable ordinal is in* $\mathsf{S}$*)*. **We use transfinite induction on the structure of ordinals to prove that all countable ordinals are elements of the set** $\mathsf{S}$**. Ordinals are characterised by the three rules** $\varnothing\in ON$**,** $\alpha\in ON\Rightarrow\alpha^+\in ON$**, and** $\bigcup_{\alpha<\lambda}\alpha\in ON$**, where** $\lambda$ **is a limit point. Proofs for the first two cases are straightforward. For the third case, say that** $\lambda=\bigcup\alpha$ **is a countable ordinal and that (by the induction hypothesis) every** $\alpha$ **is a countable ordinal that is an element of set** $CC$**. Since** $\lambda$ **is countable, there are countably many initial segments seg** $\alpha$**, and** $\lambda$ **can be constructed from the countable union of initial segments and so is also an element of the set** $\mathsf{S}$**.** □

*Theorem 3.6.* **The denotation of all MH machines is a countable computation; there is no countable computation that is not the denotation of some MH machine.**

*Proof*. **Straightforward, from definition 1.3 and lemma 3.5.** □

**We have shown that MH machines correspond to countable ordinal computation. There is a good reason to consider only countable ordinal computations, rather than ordinal computations in general. As proved below (theorem 3.7), even though machines might shrink to be arbitrarily small, they could never shrink small enough to fit an uncountable number of components, and could never run fast enough to fit an uncountable number of steps in a finite time. This argument is similar to that given by Hogarth [Hog96, p.105]; the difference is that the result here is an abstract one independent of any consideration of physics or space-time.**

> *Theorem 3.7.* **Suppose there exists a real number $r$ such that $\sum_{x\in X} k_x < r$ for some set X. Then, either X is finite or there exists a countable subset $Y\subseteq X$ such that $x\notin Y \Rightarrow k_x=0$.**
>
> *Proof*. **Let $Y_n = \{x: 1/(n+1) \le k_x < 1/n\}$. Then $Y_n$ are disjoint and finite. And, $\{x: k_x>0\} = \cup_n Y_n$, which is countable.** □

## Summary and further research

**In this chapter we restricted attention to sequential deterministic computations. A computation was defined by the underlying principle that at any stage, out of the collection of instructions still to be executed, there exists a unique instruction to be performed next. This implies that all computations have an ordinal structure (theorem 3.2). The argument is entirely abstract, independent of any properties of space or time.**

**Theorem 3.7 implies that no uncountable computation can ever be completed in a finite time. It therefore seems reasonable to restrict attention to countable computations. Theorem 3.6 implies that MH machines fully characterise all possible countable computations. This leads to an answer to the epistemological question posed at the start of the dissertation: the problems that are solvable by any machine are precisely those that can be solved by the machines belonging to the formalism presented in the first chapter.**

**However, the results of the second chapter indicate that there is no limit to the computations that can be performed by the machines in the formalism unless attention is restricted to those machines with a finite description. Different systems of finite description, moreover, lead to different computational abilities; and there appears to be no reason to choose any one system over the others.**

**This inability to find any non-arbitrary criteria for choosing a system suggest an immediate direction for further research. It is important to describe precisely which philosophical questions the existence of MH machines is intended to address—that is, we must define what is meant by 'computable'. Does the term refer to things that may be computable by some machine that simply exists? Or must the machines have a finite description? Or must some agent be able to build and program the machines? Or does it refer to the abstract principle of computation introduced in this chapter?**

**The formalisms represents a rich and interesting mathematical structure that also gives rise to further technical questions. First, although non-deterministic Turing computation is no more powerful than deterministic Turing computation, it might be that non-deterministic MH machines are more powerful than deterministic ones. Second, while formalisms with an uncountable number of states may be physically implausible, it would be interesting to explore their properties. Third, it would be interesting to extend the formalism to cover the case of computations with an uncountable ordinal denotation. Fourth, it is not yet clear how the formalism in this dissertation machines relate to other approaches to transfinite computation such Turing's work [Tur39] on transfinite logics.**

# BIBLIOGRAPHY


**[Chu36]** *An Unsolvable Problem of Elementary Number Theory*. **Church, A. 1936. American Journal of Mathematics 58:345–363.**

**[Dal78]** *Sets: Naive, Axiomatic and Applied*. **van Dalen D., Doets H.C., de Swart H.**

**[Del70]** *A Profile of Mathematical Logic*. **Delong, H. 1970. London: Addison-Wesley.**

**[Dav65]** *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems, and Computable Functions*. **ed. Davis, M. 1965. New York: Raven Press.**

**[Ear94]** *Infinite Pains: The Trouble with Supertasks*. **Earman, J., Norton, J. 1994. In "Paul Benacerraf: The Philosopher and His Critics" ed. S. Stick. New York: Blackwell.**

*On formally undecidable propositions of principia mathematica and related systems*. **trans. B. Meltzer 1962 Oliver & Boyd: London. Translation of "Uber formal unentschiedbare Sätze der Principia Mathematica und verwandter System I" by Kurt Gödel. in Monatshefte für Mathematik und Phsyic vol. 38. pp. 173-198. Leipzig: 1931.**

**[Göd34]** *On undecidable propositions of formal mathematical systems*. **Gödel K 1934. Lecture notes, Institute for Advanced Study, with corrections. Reprinted in [Dav65].**

**[Flo67]** *Assigning meanings to programs*. **Floyd R.W. 1967. In "Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19 (American Mathematical Society), Providence, pp. 19-32.**

**[For94]** *An Easy Induction: Introductory Notes on Induction and Recursion*. **Forster T. 1994. DPMMS, University of Cambridge.**

**[Hal60]** *Naive Set Theory*. **Halmos P.R. 1960. New York: Springer-Verlag.**

**[Hog92]** *Does General Relativity Allow An Observer To View An Eternity In A Finite Time?* **Hogarth, M. 1992. Foundations of Physics Letters 5.2:173–181.**

**[Hog94]** *Non-Turing Computers And Non-Turing Computability*. **Hogarth, M. 1994. PSA I:126–138. Not much of a paper, but readable.**

**[Hog96]** *Predictability, Computability and Spacetime*. **Hogarth, M. 1996. PhD Thesis, University of Cambridge, Ch. 3.**

**[Joh93]** *Notes on logic and set theory*. **Johnstone P.T. 1987. Cambridge: University Press**

**[Kle67]** *Mathematical Logic*. **Kleene, S.C. 1967. New York: John Wiley. Apparently has a good introduction to partial and primitive recursive theory. Haven't read it.**

**[Min67]** *Computations: finite and infinite machines*. **Minsky M. 1967. London: Prentice Hall.**

**[Pot90]** *Sets: an introduction*. **Potter M.D. 1990. Oxford: Clarendon Press.**

**[Tur37]** *On Computable Numbers, With An Application To The Entscheidungsproblem*. **Turing, A.M. 1937. Proceedings of the London Mathematical Society 42:230–265; a correction 43:544–546. The Original Paper From Hallowed-Be-His-Name-Turing. The waffly stuff is good. But the equations are yucky. DeLong's version is faithful to the notation but easier to understand.**

**[Tur39]** *Systems of logic based on ordinals*. **Turing, A.M. 1939. Proceedings of the London Mathematical Society 2, 45:161-228. Reprinted in [Dav65].**

**[Wis96]** *A generalisation of the halting problem to non-finite computation*. **Wischik, L.J. 1996. M.Phil. essay, Dept. of H.P.S., University of Cambridge.**

**[Zuk74]** *Sets and transfinite numbers*. **Zuckerman M.M. 1974. Macmillan: London.**