Infinite time computations and infinite algorithms

Anton Broberg

Master's Essay in Logic
December 18, 2008
Department of Philosophy
University of Gothenburg
Supervisor: Fredrik Engström

Contents:

Infinite time computations and infinite algorithms

Anton Broberg

Abstract

In this paper we investigate infinite time Turing machines as defined by Hamkins and Lewis in [1]. We extend the result in [2] showing that a larger set of clockable ordinals are 1-tape clockable. Furthermore, a new notion of infinite time Turing machine, in which the set of states may be infinite, is compared with the original notion. We show that the strength of such infinite state infinite time machines correspond to the strength of infinite time Turing machines equipped with real-oracles.

1. Introduction

The notion of infinite time Turing machines (defined in section 2) where introduced in [1] by Hamkins and Lewis. Most of the results in section 2,3,5 and 6 are from this article. Some of these results have (more or less) alternative proofs, where the new proof of the Lost Melody Theorem (Theorem 5.11) might be of most interest since the original proof is quite complicated.

Welch shows in [3] that every clockable ordinal is writable (these notions are to be defined in section 3), which was an open question in [1]. The proof is examined in section 4.

In [2], Hamkins and Seabold defines infinite time Turing machines with only one tape (in the original definition the machines have three tapes) and shows that there is clockable ordinals that are not 1-tape-clockable (clockable by a machine with one tape). It is however not known exactly which clockable ordinals that are 1-tape-clockable. This is investigated in section 7 (where the results up until Definition 7.5 are, with some modifications, from [2]).

The main goal of this paper is to investigate what would happen if we where to allow the infinite time Turing machines to have an infinite amount of states. A number of questions arises. How 'powerful' will these machines be? If the infinite set of states of such a machine is recursive, will the resulting function this machine computes be 'ordinary' infinite time computable? Will these machines be able to solve the halting problems for infinite time machines? These infinite state infinite time machines are defined and examined in section 8.

We will assume some basic knowledge about recursion theory and basic ordinal arithmetics.

Every machine in this paper is fictional and any resemblance with a real machine is coincidental.

2. Infinite time Turing machines (IT-machines)

An IT-machine works like a Turing machine that can keep on computing for a transfinite ordinal length of time. The machine has three tapes (an input-tape, a scratch-tape and an output-tape) with countably many cells on each, ordered as ω. It has a head that moves along the three tapes, one cell at a time, reading and writing 1's and 0's from/to the cells. The head starts every computation to the far left at the beginning of the tapes.

When the head reads, it reads three cells (one from each tape) at once, but it can only write to one cell at a time. It would make no difference in the computing power (in the sense that the same functions would be computable) if we allowed the machines to read only from one cell at a time, but it would make a difference in the possible lengths of certain computations (more on this in section 7).

A program is a finite set of tuples $\langle s, r, c, s' \rangle$ that specifies the machines actions, that is, when the machine is in state s and reads r from the tapes, it executes command c and enters state s'.

The possible commands that the machine can execute is to write 1 to one of the three cells the head is currently on, write 0 to one of the three cells the head is currently on, move the head one step (one cell) to the right on the tapes, move the head one step to the left[1] and finally the machine can halt (that is, end the computation). When the machine executes one of these commands (except for when it halts) we say that the machine has made a step of computation. The length of a computation is the order type of the steps of computation the machine makes before it halts.

After α steps of computation, where α is a limit-ordinal the machine enters a limit-stage. Every cell on the tapes that changes value cofinally often before α is set to 1, and the other cells, that have stabilized, remains the same. The head is instantly placed on the first cells of the tapes and in the program, a special limit-state is consulted.

There are two ways for a machine to halt. Either the machine is told to halt by the program, or there is no tuple in the program that corresponds to the machines current setup.

For convenience, let the set of all possible states be ω. Let 1 be the start-state for all programs (so a program that does not contain a tuple with the state 1, is simply the program that halts after 0 steps of computation) and let 0 be the limit-state. The set of possible reads is finite (since there is only $2^3 = 8$ different setups for the three cells the head is reading) and the set of possible commands is also finite so there is an easy recursive way of coding the set of all possible tuples to ω. In this way we can code every program with a finite subset of ω (and every finite subset of ω codes a program). We fix a recursive bijection between ω and the finite subsets of ω, thus every $n \in \omega$ codes a program (and every program is coded by an $n \in \omega$). This allows us to be a little careless and call a natural number p a program when we really mean the program that p codes.

The machine can be told to do different things at once, for example: both $\langle s, r, c_0, s'_0 \rangle$ and $\langle s, r, c_1, s'_1 \rangle$ may be elements of a program, but then the machine chooses to obey the tuple with the least natural number coding it.

[1] If a machine is told by the program to move the head a step to the left when the head is placed at the beginning of the tapes, the machine simply ignores the command.

At the start of the computation the input is written to the input-tape, and the scratch-tape and the output-tape are filled with 0's.

The inputs and outputs for the machines are countable binary strings. These binary strings will be referred to as reals.

Every program p determines a partial function φ_p: $2^\omega \to 2^\omega$ simply by letting the value of $\varphi_p(x)$ be what is on the output-tape when the machine with program p and input x halts, and letting $\varphi_p(x)$ be undefined (which will be denoted $\varphi_p(x)\uparrow$) when the machine does not halt. Of course $\varphi_p = \varphi_q$ iff $\varphi_p(x)\downarrow \leftrightarrow \varphi_q(x)\downarrow$ for all x and $\varphi_p(x) = \varphi_q(x)$ for all x such that $\varphi_p(x)\downarrow$.

By adding extra input-tapes we get functions of many variables as well.

A (partial) function f on reals is IT-computable if there is a program $p \in \omega$ such that $\varphi_p = f$ (that is $\varphi_p(x) = f(x)$ if $f(x)$ defined and $\varphi_p(x)\uparrow$ otherwise).

A set of reals A is IT-decidable if its characteristic function is IT-computable, and A is IT-semi-decidable if there is an IT-computable function f such that $f(x) = 1$ iff $x \in A$.

We will regard the natural numbers as a subset of the reals by letting the real where the first n digits are 1 and the rest is 0, represent the natural number n.

From now on, 'decidable', 'computable' and so forth will mean IT-decidable and IT-computable while 'recursive' will refer to ordinary recursiveness in the Turing sense.

THEOREM 2.1: For every computable function φ_p such that φ_p is defined exactly on ω, $\varphi_p(n)$ is not computable in less than ω steps of computation for any $n \in \omega$.

Proof: Assume the machine with program p and input $n \in \omega$ halts in k steps of computation for some $k \in \omega$. In k steps of computation, the head has not time to move more than at most k steps to the right on the tapes. Thus, if we run this program with an input x that is any real where the first $k+1$ digits in x are like the first $k+1$ digits in n, this program will halt in k steps on this input as well. But this is a contradiction since φ_p is defined only on ω. $\dashv$

REMARK 2.2: If the tapes look exactly the same in two limit-stages and none of the cells that are 0 in these limit-stages, turns 1 at any point between them, then the machine is caught in a loop. Since the head always is in the same place in limit-stages and the program always consults the limit-state in limit-stages, the exact same thing will happen again. But if any of the cells that are 0 turns 1 at some point between the two limit-stages, then the machine could escape the loop in a later limit-of-limits where this cell then turns 1 (for an example of such an escape, see (6) on page 6).

THEOREM 2.3: Every halting IT-computation is countable.

Proof: Suppose a computation has not halted after ω_1 steps of computation. Let α_0 be the first stage where all the cells that locally stabilize before ω_1 has already stabilized. Since there are only countably many such cells, we have that $\alpha_0 < \omega_1$ (α_0 is the supremum of a countable set of countable ordinals and therefor countable). The cells that change value at all between α_0 and ω_1 will change value cofinally often between α_0 and ω_1, so there must be a countable sequence $\alpha_0 < \alpha_1 < \alpha_2 < ...$

of countable ordinals such that all the cells that change at all after α_n have changed value at least once before α_{n+1}. But then stage $\delta = \sup\{\alpha_n\}$ is a countable limit-stage where the tapes looks exactly like they do at stage ω_1 and none of the cells that are 0 at stage δ will turn 1 again before ω_1. This computation is therefore caught in a loop (according to Remark 2.2) and will never halt. ⊣

Since we in this proof can choose α_0 as big as we want, as long as it is countable, we can choose α_0 to be bigger than δ and we would produce another $\delta' > \delta$ that is a countable limit-stage where the tapes look exactly like in stages δ and ω_1. Hence, we can draw the following conclusion.

COROLLARY 2.4: Every non-halting computation is caught in a loop, and the length of this loop is countable.

THEOREM 2.5: Every recursively enumerable set $A \subseteq \omega$ is decidable.

Proof: We specify the program p that decides A. The program is set up so that if there is a 1 on the first cell of the scratch-tape in a limit-stage, the machine halts with output 'no'. On input x, p first decides if x is a natural number or not. p does this by first writing a 1 to the first cell of the scratch-tape and then it starts searching for the first 0 on the input-tape. If there is no first 0, that is, if the input-tape is filled with 1's, the search will be in vain and the machine will, after ω steps of computation, be in a limit-stage with a 1 on the first cell of the scratch-tape, and the machine will halt with output 'no'. If it finds a 0 it moves back to the first cell and writes a 0 to the scratch-tape. Then it returns to the cell where it read the first 0 and starts a search for the next 1. If another 1 is found, x is not a natural number and we halt with output 'no', but if there is no 1 (after the first 0), the machine will, after ω steps of computation, be in a limit-stage with a 0 on the first cell of the scratch-tape. And it is easy to see that this happens exactly when x is a natural number.

Thus, if the machine has a 0 on the scratch-tape in the first limit-stage, this means that the input x is a natural number, and it is time for the machine to decide if $x \in A$ or not. Since A is recursively enumerable, there is a Turing machine that halts iff the input is a member of A. So now the machine first writes a 1 to the first cell of the scratch-tape and then simply simulates this Turing machine on the input-tape where the input x still is untouched. If the simulated machine halts, then p halts, since this implies that $x \in A$. If the simulated machine does not halt, the machine will, after another ω steps of computation, be in a limit-stage with a 1 on the scratch-tape and consequently, the machine halts with output 'no'. ⊣

COROLLARY 2.6: The halting problem for Turing machines is solvable by an IT-machine.

Proof: Follows directly since $K = \{n;$ the n'th Turing machine halts on input $n\}$ is recursively enumerable. ⊣

Every relation $\lhd \subseteq \omega \times \omega$ is coded by some real x by letting the $\langle n, m\rangle$'th digit in x be 1 iff $n \lhd m$ (where $\langle\ ,\ \rangle$ is some computable pairing function). It is easy to see that every real codes a binary relation in this way and every binary relation is coded by a real.

Let WO be the set of reals that codes well orders and for all $x \in$ WO let $\|x\|$ be the order type of the well order that x codes, hence if $x \in$ WO, then $\|x\|$ is a countable ordinal.

WO is Π^1_1-complete, which means that WO is Π^1_1 and if A is Π^1_1, then there is a recursive function f on the reals such that $x \in A$ iff $f(x) \in$ WO.[2]

In [1], Hamkins and Lewis shows that WO is IT-decidable and since WO is Π^1_1-complete, this implies that all Π^1_1- and subsequently also all Σ^1_1-sets are decidable. It is also shown that the power of the IT-machines extends further by showing that some Δ^1_2-sets (that are not Π^1_1- or Σ^1_1-sets) are decidable. But this is where it stops. All decidable and semi-decidable sets are Δ^1_2 (for details on this, see [1]).

THEOREM 2.7 (S-N-M): There is a recursive function s such that $\varphi_p(\overline{k}, \overline{x}) = \varphi_{s(p,\overline{k})}(\overline{x})$, for all reals $\overline{x}$ and natural numbers p and $\overline{k}$.

The classical proof of the s-n-m theorem works in this context as well.

Notice that s is defined only on ω and hence the k_i's in $\overline{k}$ must not be any reals but natural numbers. If we let go of this restriction we get in a lot of trouble. Assume there is a computable function s such that $\varphi_p(\overline{k}, \overline{x}) = \varphi_{s(p,\overline{k})}(\overline{x})$, for all reals $\overline{x}$ and $\overline{k}$ and natural numbers p, and consider the computable function φ_p such that $\varphi_p(x, y) = x$ for all reals x and y. Then we have that $\varphi_{s(p,x)}(y) = x$. But then $\varphi_{s(p,x)}(0) = x$. This means that for any reals x and x' such that $x \neq x'$, we have that $\varphi_{s(p,x)} \neq \varphi_{s(p,x')}$ which would imply that there is at least $2^{\aleph_0}$ many computable functions and that is a contradiction.

THEOREM 2.8 (RECURSION): For any IT-computable total $f\colon \omega \to \omega$, there is a program p such that $\varphi_p = \varphi_{f(p)}$.

Again we omit the proof since the classical proof goes through.

3. Clockable and writable ordinals

DEFINITION 3.1: An ordinal α is *clockable* if there is an IT-machine that halts after α steps of computation on input 0. Let ClO be the set of clockable ordinals.

DEFINITION 3.2A: A real x is *writable* if there is an IT-machine that halts with output x on input 0. Let WrR be the set of writable reals.
DEFINITION 3.2B: An ordinal α is writable if there is a real $x \in \text{WrR} \cap \text{WO}$ such that $\|x\| = \alpha$. Let WrO be the set of writable ordinals.

[2] A is Π^1_1 if there is a second-order arithmetical formula $\forall \overline{X} \psi(\overline{X}, Y)$, where ψ is a first-order formula, that defines A in the standard model.

So which ordinals are clockable? Let us begin by recognizing some simple clockable ordinals:

(1) $n \in \mathrm{ClO}$ for all $n \in \omega$. (Consider a program with n states.)
(2) $\omega \in \mathrm{ClO}$. (Let the limit-state halt the machine.)
(3) $\alpha \in \mathrm{ClO} \Rightarrow \alpha + 1 \in \mathrm{ClO}$. (Just add an extra state to the program and instead of halting, go one step to the right and enter this new state and halt.)
(4) $\alpha \in \mathrm{ClO} \Rightarrow \alpha + \omega \in \mathrm{ClO}$. (Move the computation one cell to the right and leave the first cell untouched. In the limit-state check if the first cell is 1, if so halt, otherwise continue with the computation. Instead of halting when the original machine halts, move to the first cell and write 1. Then enter a new state that keeps the head stepping to the right. In the next limit-stage, when the program consults the limit-state and reads a 1 from the tape the machine will halt. This will take $\alpha + n + \omega = \alpha + \omega$ steps.)
(5) $\alpha \in \mathrm{ClO} \Rightarrow \alpha + \beta \in \mathrm{ClO}$ for all $\beta < \omega^2$. (Follows from (3) and (4) since every $\beta < \omega^2$ can be written on the form $\omega \cdot n + k$ for some natural numbers n and k (Cantor's normal form).)
(6) $\omega^2 \in \mathrm{ClO}$. (Consider the machine that only steps to the right and in every limit first writes 1 to the first cell and then writes it over with a 0 (this will be referred to as 'flashing a flag'). Since ω^2 is the first limit of limits, this will be the first time the first cell is 1 in a limit-stage. This will be noticed by the program and the machine will halt.)

THEOREM 3.3: Every recursive ordinal is clockable.

Proof: Cantor's normal form implies that every ordinal can be written on the form $\alpha + \beta$ where α is a limit of limits (or 0) and $\beta < \omega^2$, hence (5) above implies that it suffices to show that every recursive limit of limits ordinal is clockable. The proof of this is described in great detail in [1] and will only be sketched here. Let α be a recursive limit of limits ordinal. Then there is a real x such that x is writable in ω many steps and $\|x\| = \alpha$. So we construct a machine that first writes x and then erases the smallest element in the order-relation that x codes until the relation is empty, then it stops. This machine halts after $\omega + (\omega + \omega)\alpha + \omega$ steps (first ω steps to write x, then for every element in the relation that x codes, it will take ω steps to find the smallest element and ω steps to erase all mention of this element, then it will take ω steps to realize that the relation is empty). Then we use that α is a limit of limits to shorten this algorithm to the extent that it actually will be done in α many steps (see [1] for details). Hence $\alpha \in \mathrm{ClO}$. ⊣

THEOREM 3.4 (SPEED-UP LEMMA): If $\alpha + n \in \mathrm{ClO}$, then $\alpha \in \mathrm{ClO}$ for all α and all $n \in \omega$.

Proof: We may assume that α is a limit ordinal, since it is easy to make a computation a finite amount of steps longer. Consider the program p that halts after $\alpha + n$ steps of computation. In the limit-stage α the first $n + 1$ cells on the tapes must be set up in such a way that the machine will halt n steps from there. Denote these $n + 1$ cells with the tuple $\overline{a}$. Now consider the program that simulates p but moves the computation one step to the right, leaving the first cell on every tape

untouched. This program halts after $\alpha + n + 1$ steps of computation, and α is the first limit-stage where cells 2 to $n + 2$ on each tape are set up like $\overline{a}$. But we can see this already in the limit by flashing a flag every time any cell that is 0 in $\overline{a}$ is not 0, and by flashing a second flag every time every cell that is 1 in $\overline{a}$ has been 1 at some point since the last time we flashed the second flag. Then the first flag is 0 and the second flag is 1 in a limit-stage if and only if cells 2 to $n + 2$ on the tapes are set up as $\overline{a}$, and since the head can see three cells at once, and the first cell on every tape is free to use as flags, it is no problem to see the two flags in the limit and halt. Hence $\alpha \in \mathrm{ClO}$. $\dashv$

This proof uses the fact that the machines can read from more than one cell at a time, and in fact the Speed-up Lemma does not hold if we only allow the machines to read from one tape at a time (see section 7).

Many of the proofs ahead will use some variation of a universal machine that simulates all computations on the form $\varphi_p(0)$. By slicing up one of the tapes in ω parts and then slice up every one of those slices in three, a universal machine can actually do this on one single tape or even a slice of a tape (one have to do some modifications in what constitutes a step of computation in the simulated computations (see Remark 7.10)). Sometimes we only need the universal machine to simulate, in ω many steps, one step of computation in every simulated computation but often we need the universal machine to simulate, in ω many steps, ω steps of computation in every simulated computation. This can be done by simulating things in a certain order, for example, consider the recursive bijection from ω to $\omega \times \omega$ that orders $\omega \times \omega$ like this: $\langle 0,0\rangle$, $\langle 0,1\rangle$, $\langle 1,0\rangle$, $\langle 0,2\rangle$, $\langle 1,1\rangle$, $\langle 2,0\rangle$, $\langle 0,3\rangle$, $\langle 1,2\rangle$, $\langle 2,1\rangle$, $\langle 3,0\rangle$,... where $\langle p,k\rangle$ represents the k'th step in the computation of program p on input 0.

THEOREM 3.5: There are gaps in the clockable ordinals, and the first gap after any clockable ordinal has size ω.

Proof: Let $\alpha \in \mathrm{ClO}$ and let β be the first non-clockable ordinal greater than α. It is easy to see that β must be a limit-ordinal, and the Speed-up Lemma implies that $\beta + n \notin \mathrm{ClO}$ for all $n \in \omega$. We show that $\beta + \omega \in \mathrm{ClO}$. Consider a universal machine that on input 0 simulates all computations on the form $\varphi_p(0)$ and makes ω steps of computation of every simulated computation in ω steps. It keeps track on one of the computations that halts in α many steps, and when this simulated computation halts, it starts to flash a flag every time a simulated computation halts. Then it simply halts in the first limit-stage where this flag is 0. Since β is the first non-clockable ordinal greater than α, this flag will be 1 in every limit-stage between α and β. And it will be 1 in stage β too since the flag will be flashed cofinally often before β. But for no $n \in \omega$ will the flag be flashed at stage $\beta + n$ so in stage $\beta + \omega$ will this flag be 0. Hence, this universal machine halts in $\beta + \omega$ many steps of computation so $\beta + \omega \in \mathrm{ClO}$ which proves the theorem. $\dashv$

It is shown in [1] that the first non-clockable ordinal is the supremum of the recursive ordinals ω_1^{CK}, and in fact, for every $A \subseteq \omega$ we have that the supremum of the A-recursive ordinals is not clockable.[3]

[3]This follows from a theorem in [1] stating that no admissible ordinal is clockable.

THEOREM 3.6: For every $\alpha \in \mathrm{ClO}$, there is a gap of size at least α in the clockable ordinals.

Proof: To prove this theorem we will specify a universal machine that would halt after the supremum of the clockable ordinals if no gap of size at least α existed. The Speed-up Lemma implies that all gaps are of limit ordinal length so we can assume that α is a limit ordinal $\geq \omega \cdot 2$. Simply simulate all computations on the form $\varphi_p(0)$. In every limit, start clocking α until a simulated computation halts. Halt when the computation that clocks α halts. This occurs the first time it has been α steps since the last simulated computation halted. So if there are no gaps of size α, this computation will halt after the supremum of the clockable ordinals. (Some details are omitted here. For example would we need a flag so the machine knows if it should start over the α-clock in the limit-stages or not. There are also some technical problems with the α-clock that are addressed in Remark 7.7) $\dashv$

Since the first gap after any clockable ordinal is of size ω, the gaps of size greater than ω must start with an ordinal that is a limit of ordinals that starts gaps of size ω.

THEOREM 3.7: If $\alpha \in \mathrm{ClO} \cup \mathrm{WrO}$, the order type of the gaps of size at least α is neither clockable nor writable.

The proof of Theorem 3.7 is basically a generalization of the proof of Theorem 3.6 and is left out.

DEFINITION 3.8: A real is *eventually writable* if it is written to the output-tape of a (halting or non-halting) computation on input 0, and never is overwritten or altered. Let EWrR be the set of eventually writable reals.

An ordinal α is eventually writable if $\alpha \in \mathrm{EWrO} = \{\|x\| \,;\, x \in \mathrm{WO} \cap \mathrm{EWrR}\}$

DEFINITION 3.9: A real is *accidentally writable* if it at some point of a (halting or non-halting) computation on input 0 appears on one of the tapes. Let AWrR be the set of accidentally writable reals.

An ordinal α is accidentally writable if $\alpha \in \mathrm{AWrO} = \{\|x\| \,;\, x \in \mathrm{WO} \cap \mathrm{AWrR}\}$

Let $\lambda = \sup \mathrm{WrO}$, $\zeta = \sup \mathrm{EWrO}$ and $\Sigma = \sup \mathrm{AWrO}$. It follows practically directly from the definitions that $\mathrm{WrR} \subseteq \mathrm{EWrR} \subseteq \mathrm{AWrR}$, and subsequently also that $\mathrm{WrO} \subseteq \mathrm{EWrO} \subseteq \mathrm{AWrO}$, and subsequently $\lambda \leq \zeta \leq \Sigma$. We shall see later that in fact $\lambda < \zeta < \Sigma$, and hence all the inclusions above are strict.

THEOREM 3.10: There are no gaps in the writable ordinals.

Proof: Assume $\alpha \in \mathrm{WrO}$. We show that an arbitrary $\beta < \alpha$ also is writable. Since there is a real $x \in \mathrm{WrR}$ such that x codes a well order $\lhd$ and $\|x\| = \alpha$, there must be an $n \in \omega$ such that if a real y codes the relation we get if we erase all mentioning of any m such that $n \lhd m$ out of the relation that x codes, then $\|y\| = \beta$. So since y is writable (just write x and start erasing), we have that $\beta \in \mathrm{WrO}$. $\dashv$

THEOREM 3.11: The order types of ClO and WrO are both equal to λ.

Proof: The order type of WrO is λ since there are no gaps in WrO. First we show that the order type of ClO is greater than or equal to λ by showing that for every $\alpha \in$ WrO, there are at least α many[4] clockable ordinals.

Fix an $x \in$ WrR such that $\|x\| = \alpha$. For every $n \in \omega$, consider the programs p_n that writes up x on a tape and then starts erasing the $\lhd$-least element in the relation that x codes until n is the $\lhd$-least element in the relation. For every $n \in \omega$, let α_n be the ordinal clocked by p_n. Then for every $n, m \in \omega$ such that $n \lhd m$, we have that $\alpha_n < \alpha_m$. Hence, the order type of the clockable ordinals generated by these programs is α.

Now for the other direction we show that the order type of all clockable ordinals less than an arbitrary $\alpha \in$ ClO, is writable. Consider a universal machine that simulates all the computations on the form $\varphi_p(0)$ and at the same time clocks α, in such a way that one step of computation is made in every $\varphi_p(0)$ and in the α-clock, in ω many steps. We will make sure that this machine writes a real x, that codes a relation $\lhd$ such that $\|x\|$ is of the wanted order type, at the output-tape. At the start and after every limit-stage, it first makes one step in the α-clock and then it makes one step in $\varphi_0(0), \varphi_1(0), \varphi_2(0)...$ and so forth, and as soon as a computation p halts, it will add p to the relation $\lhd$ by writing a 1 to the $\langle q, p\rangle$'th cell on the output-tape for every q that has been added earlier. And then nothing more is done to the output-tape until the next limit-stage is reached, regardless if more simulated computations halt. When the α-clock halts, we halt with a real x written to the output-tape that codes a relation $\lhd$ such that $p \lhd q$ if the program p halts before q on input 0, and both halts before α. Since only the 'least' program that halts in a certain amount of steps are members of the relation, x will actually code a well order, and in fact $\|x\|$ is equal to the order type β of all clockable ordinals smaller that α, and consequently $\beta \in$ WrO. $\dashv$

THEOREM 3.12: $\lambda \notin \text{ClO} \cup \text{WrO}$ but $\lambda \in$ EWrO.

Proof: To see that $\lambda \in$ EWrO, just run the same algorithm as in the second part of the proof of Theorem 3.11, but without the α-clock. That program would run forever and eventually write λ to the output-tape.

Of course $\lambda \notin$ WrO, but to see that $\lambda \notin$ ClO, assume that $\lambda \in$ ClO and simulate all computations on the form $\varphi_p(0)$ and every time a computation halts, make one step of the computation that clocks λ. When this λ-clock halts, we halt. But since the order type of ClO is λ, this computation will halt after the supremum of the clockable ordinals. That is a contradiction. $\dashv$

THEOREM 3.13: $\lambda < \zeta < \Sigma$.

Proof: That $\lambda < \zeta$ follows directly from 3.12. To see that $\zeta < \Sigma$, consider the universal machine that simulates all computations on the form $\varphi_p(0)$ on the scratch-tape, and after it has made one step of computation on every simulated computation it adds up all the ordinals that are written to the simulated output-tapes (to see

[4]What we really mean when we write that 'there are α *many* clockable ordinals', is that the order type of these ordinals is α. We will misuse the word 'many' in this way throughout this paper.

that this is a computable procedure, see Remark 3.14) and writes the resulting ordinal to the real output-tape. Eventually, all the eventually writable ordinals will be written to the output-tapes of the simulated computations, and after that point, the ordinals written to the real output-tape will always be greater or equal to ζ. Since these ordinals are written to a tape, they are accidentally writable and consequently strictly smaller than Σ. Hence $\zeta < \Sigma$. $\dashv$

REMARK 3.14: It is no problem to check if a real on a simulated output-tape codes an ordinal or not so assume this is done and we have ω ordinals on slices of the scratch-tape. Let $z_0, z_1, ...$ be the reals coding these ordinals. We now define a relation $\lhd$ on ω such that $\langle n, k\rangle \lhd \langle n', k'\rangle$ iff k is related to something in the well-order z_n codes and k' is related to something in the well-order $z_{n'}$ codes and either $n < n'$ or $n = n'$ and $\langle k, k'\rangle$ is a member of the well-order that z_n codes. This relation is a well-order and the order type is equal to $\|z_0\| + \|z_1\| + ...$, and this relation is decidable (with access to $z_0, z_1, ...$) so the real coding it is writable (with access to $z_0, z_1, ...$) and this real is what we write to the output-tape in Theorem 3.13.

4. ClO $\subseteq$ WrO

An open question in Hamkins and Lewis's article ([1]) was whether ClO $\subseteq$ WrO or not. This was answered affirmative by Philip Welch in [3].

Assume that we have ordered the cells on the three tapes as ω in some computable way.

DEFINITION 4.1: For every $i, p \in \omega$ and every ordinal α, $C_i^p(\alpha)$ is the value (0 or 1) of the i'th cell after α steps of computation with program p on input 0.

DEFINITION 4.2: $\delta_i^p(\alpha) = \sup\{\beta < \alpha;\ \beta = 0 \vee C_i^p(\beta) \neq C_i^p(\beta + 1)\}$, for every $i, p \in \omega$ and every ordinal α.

Hence $\delta_i^p(\alpha)$ is the last stage, before α, that the i'th cell changed value in the computation of $\varphi_p(0)$. It is easy to see that when α is a limit ordinal, we have that $\delta_i^p(\alpha) = \alpha$ iff the i'th cell changed value cofinally often before α (or equivalently $\delta_i^p(\alpha) \neq \alpha$ iff $\delta_i^p(\alpha) < \alpha$ iff the i'th cell has stabilized before α). Of course, if $\delta_i^p(\alpha) < \alpha$ and the i'th cell has stabilized before α, it might still 'destabilize' at some stage after α.

THEOREM 4.3: $\delta_i^q(\Sigma) < \Sigma \ \Rightarrow\ \delta_i^q(\Sigma) < \zeta$, for every $i, q \in \omega$.

Proof: Theorem 4.3 is proved in [3] and will not be rigorously proved here. In words it states that if a cell i in a computation (on input 0) has (locally) stabilized before Σ, it actually stabilized already before ζ. Welch proves this by constructing a non-halting universal machine $M(q, i)$ that, if $\delta_i^q(\Sigma) < \Sigma$, eventually writes $\delta_i^q(\Sigma)$ to the output-tape, and never writes it over. Hence, the conclusion that $\delta_i^q(\Sigma) \in$ EWrO, can be drawn, which proves the theorem.

The way that $M(q, i)$ does this is that after every time it has made one step of computation on every simulated computation on the form $\varphi_p(0)$, it computes the sum σ of all the ordinals that are written to any of the simulated computations

simulated tapes (this can be done according to Remark 3.14), then it computes $\delta_i^q(\sigma)$. Let x be what is currently on the output-tape at this point. Now the machine checks if $x \in \mathrm{WO}$, if $\delta_i^q(\sigma) \leq \|x\|$ and if $\delta_i^q(\|x\|) = \|x\|$. If x withstands all these tests, the output-tape is left as it is, otherwise (a code for) $\delta_i^q(\sigma)$ is written to the output-tape instead of x.

Now assume that $\delta_i^q(\Sigma) < \Sigma$. We show that $\delta_i^q(\Sigma) < \zeta$. First we show that $\delta_i^q(\Sigma)$ will be written to the output-tape of $M(q,i)$ at some point of the computation. At some point will an accidentally writable ordinal that is greater than $\delta_i^q(\Sigma)$ be written to a simulated tape, so the sum σ will at this point be such that $\delta_i^q(\Sigma) < \sigma < \Sigma$, and consequently will $\delta_i^q(\Sigma) = \delta_i^q(\sigma)$. The only way that (a code for) $\delta_i^q(\sigma) = \delta_i^q(\Sigma)$ is not written to the output-tape at this point is if $x \in \mathrm{WO}$, $\delta_i^q(\sigma) = \delta_i^q(\Sigma) \leq \|x\|$ and $\delta_i^q(\|x\|) = \|x\|$, where x is what is currently on the output-tape. But if x withstands these tests we must have that $\delta_i^q(\Sigma) = \|x\|$ since $\delta_i^q(\|x\|) = \|x\|$ and $\delta_i^q(\Sigma) \leq \|x\|$. In either case must $\delta_i^q(\Sigma)$ at some point be written to the output-tape.

It is fairly easy to check that when (a code for) $\delta_i^q(\Sigma)$ has been written to the output-tape, it will never be overwritten, by basically the same arguments as above, hence $\delta_i^q(\Sigma) \in \mathrm{EWrO}$, which proves the theorem. ⊣

STATEMENT 4.4: $\delta_i^q(\Sigma) = \Sigma \Rightarrow \delta_i^q(\zeta) = \zeta$, for every $q, i \in \omega$.

Some version of Statement 4.4 is used in [3] to prove the statements (I) and (II) below, but is never stated nor proved in [3].

For every program $p \in \omega$, we have:

(I) $C_i^p(\Sigma) = C_i^p(\zeta)$ for all $i \in \omega$.

(II) $C_i^p(\zeta) = 0$ implies that $C_i^p(\xi) = 0$ for all $\zeta < \xi < \Sigma$.

If $\delta_i^p(\Sigma) < \Sigma$, Theorem 4.3 shows that $\delta_i^p(\Sigma) < \zeta$ and (I) follows. On the other hand if $\delta_i^p(\Sigma) = \Sigma$, Statement 4.4 shows that $\delta_i^p(\zeta) = \zeta$ and hence $C_i^p(\Sigma) = C_i^p(\zeta) = 1$. Either way, (I) holds.

$C_i^p(\zeta) = 0$ implies that $\delta_i^p(\zeta) < \zeta$ which implies (according to Statement 4.4) that $\delta_i^p(\Sigma) < \Sigma$ which implies (according to Theorem 4.3) that $\delta_i^p(\Sigma) < \zeta$ which implies that $C_i^p(\xi) = 0$ for all $\zeta < \xi < \Sigma$, hence, (II) holds as well.

(I) and (II) implies that if $\varphi_p(0)$ has not halted before Σ, it is caught in a loop and will never halt (according to Remark 2.2), in other words, every halting computation (on input 0) halts before Σ.[5]

THEOREM 4.5: $\mu < \lambda$ for every $\mu \in \mathrm{ClO}$, and consequently, since there are no gaps in the writable ordinals, $\mathrm{ClO} \subseteq \mathrm{WrO}$.

Proof: Let p be a program such that $\varphi_p(0)$ halts after μ steps of computation. Then we know that $\mu < \Sigma$. Let q be a program such that $\varphi_q(0)$ at some point writes $\mu' \geq \mu$ on some of its tapes. Let N be a machine that simulates $\varphi_q(0)$ on

[5] In [3], Welch draws the conclusion that all computations on input 0 halts before ζ from (I) and (II). Unfortunately, we fail to see how this conclusion can be drawn, but it does not really matter since we (with some minor modifications) only need them to halt before Σ.

a slice of the scratch-tape, and every time an ordinal α is written to one of φ_q's simulated tapes, N writes α on the real output-tape and then runs α steps in the computation of $\varphi_p(0)$. If $\varphi_p(0)$ halts, then N halts. Eventually $\varphi_q(0)$ will write an ordinal $\overline{\mu} \geq \mu$ on the output-tape and then φ_p will have time to halt so N will halt with $\overline{\mu}$ on the output-tape, so $\overline{\mu} \in \mathrm{WrO}$. Hence $\mu \leq \overline{\mu} < \lambda$, which proves the theorem. ⊣

COROLLARY 4.6: $\sup \mathrm{ClO} = \lambda$.

Proof: For every $\alpha \in \mathrm{WrO}$ there is a $\beta \in \mathrm{ClO}$ such that $\alpha \leq \beta$. For example, β could be the ordinal clocked by the program that first writes x such that $\|x\| = \alpha$ to a tape and then starts erasing the smallest element from the relation that x codes and halts when this relation is empty. This computation will take at least α many steps before it halts. Hence, $\sup \mathrm{ClO} \geq \lambda$. But Theorem 4.5 implies that $\sup \mathrm{ClO} \leq \lambda$, hence $\sup \mathrm{ClO} = \lambda$. ⊣

Although we have no proof for Statement 4.4, we will for the rest of this paper assume that it is true, and hence that in fact $\sup \mathrm{ClO} = \lambda$.

5. The halting problems

The halting problem can be relativized to IT-machines in two different ways, $H = \{(p, x) : \varphi_p(x)\downarrow\}$ and $h = \{p : \varphi_p(0)\downarrow\}$ which are not (as their classical analogs) equivalent. Notice also that $h \subseteq \omega$, but $H \nsubseteq \omega$.

It is easy to see that H and h are semi-decidable, just simulate $\varphi_p(x)$ and if it halts, we give 'yes' as an output.

THEOREM 5.1: H and h are not decidable

Proof: Assume H is decidable. Then there is an IT-computable function r such that $r(p, x) = 1$ if $\varphi_p(x)\downarrow$ and $r(p, x) = 0$ otherwise. But then there is a program q that computes

$$\varphi_q(p) = \begin{cases} 1 & \text{if } r(p,p) = 0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

This implies that $\varphi_q(q)\downarrow$ iff $\varphi_q(q) = 1$ iff $r(q, q) = 0$ iff $\varphi_q(q)\uparrow$. That is a contradiction, hence, H is not decidable.

Now assume h is decidable. That implies that $\tilde{h} = \omega - h$ also is decidable. It is easy to see that these two sets are countable. We will now define a bijection g: $h \to \tilde{h}$ simply by letting the n'th element in h go to the n'th element in $\tilde{h}$. Since both h and $\tilde{h}$ are countable and decidable, g must be computable. Define f: $\omega \to \omega$ simply by letting $f(p) = g(p)$ if $p \in h$ and $f(p) = g^{-1}(p)$ otherwise. Since g is computable and $h \cup \tilde{h} = \omega$, f is a total computable function on ω. The recursion theorem implies that there is a program q such that $\varphi_q = \varphi_{f(q)}$ and hence $\varphi_q(0)\downarrow$ iff $\varphi_{f(q)}(0)\downarrow$. But by the definition of f we have that $\varphi_q(0)\downarrow \Rightarrow q \in h \Rightarrow f(q) = g(q) \Rightarrow f(q) \in \tilde{h} \Rightarrow f(q) \notin h \Rightarrow \varphi_{f(q)}(0)\uparrow$. This is a contradiction, hence, h is not decidable. ⊣

Let $H_\alpha = \{(p,x);\ p \text{ halts in less than } \alpha \text{ steps of computation on input } x\}$ and $h_\alpha = \{p;\ p \text{ halts in less than } \alpha \text{ steps of computation on input } 0\}$.

THEOREM 5.2: For every $\alpha < \omega_1$ there is a program p and a real x such that p halts in α many steps on input x.

Proof: Every ordinal α can be written on the form $\beta+\delta$ where β is a limit of limits ordinal (or 0) and $\delta < \omega^2$ (Cantors normal form). Let x be such that $\|x\| = \beta$ and let p be the program that first counts through x like in the proof of Theorem 3.3. This can be done in $\|x\| = \beta$ many steps since β is a limit of limits. When this is done, p simply clocks δ and halts. $\dashv$

Hence, for all countable ordinals α and β such that $\alpha \neq \beta$ we have that $H_\alpha \neq H_\beta$.

But this does not hold for the h_α's. For example, if $\alpha \notin \text{ClO}$ and β is the first clockable ordinal greater than α we have that $\alpha \neq \beta$ but $h_\alpha = h_\beta$. We also have that $h_\alpha = h$ for all $\alpha \geq \lambda$.

THEOREM 5.3: H_α and h_α are decidable for every $\alpha \in \text{WrO}$.

Proof: First we show that H_α is decidable. Consider the program that on input (p,x) first writes z, where $\|z\| = \alpha$, on a slice of the scratch-tape. This can be done since $\alpha \in \text{WrO}$. Then it starts to simulate the program p on input x, and for every step of computation in the simulation of p, it erases the least element in the well-order that z codes. If the simulation of p halts before this well-order is erased, it halts with output 'yes', and if the well-order is completely erased before p has halted, it halts with output 'no'. This decides if $(p,x) \in H_\alpha$ or not. And of course, to decide if $p \in h_\alpha$, just check if $(p,0) \in H_\alpha$, hence h_α is also decidable. $\dashv$

THEOREM 5.4: H_λ and h_λ are semi-decidable but not decidable.

Proof: That h_λ is semi-decidable but not decidable follows directly since $h_\lambda = h$. It is also easy to see that if H_λ is decidable, then so is h_λ, hence is H_λ not decidable either. So it suffices to show that H_λ is semi-decidable. But that is easy. On input (p,x), just start simulating the program p on input x, and at the same time simulate every program on input 0 (like a universal machine). If the simulation of p halts, we wait for some of the other simulated programs on input 0 to halt, and if one of them does after p has halted, we halt with output 'yes'. $\dashv$

THEOREM 5.5: For any limit ordinal $\alpha \in \text{ClO}$, H_α and h_α are not α-decidable, but they are α-semi-decidable and $(\alpha+1)$-decidable.

Proof: Consider the program q that on input (p,x) simulates the program p on input x while simultaneously clocking α. This can be done in such a way that in ω steps of computation of q, ω steps of computation will be simulated in both p and the α-clock. If p halts before the α-clock, q halts with output 'yes', but if the α-clock halts, we can arrange for q to recognize this in the α'th step of computation

and halt with output 'no'. The program q will halt in less than $\alpha+1$ steps of computation on every input and it decides H_α. Hence H_α and h_α (to decide if $p \in h_\alpha$ in less than $\alpha+1$ steps, just run q with input $(p,0)$) are $\alpha+1$-decidable. But q also shows that H_α and h_α are α-semi-decidable, since q halts with output 'yes' on input (p,x) in less than α steps iff $(p,x) \in H_\alpha$. Now it remains to show that h_α and H_α are not α-decidable. But the second part of the proof of Theorem 5.1, that shows that h is not decidable, is generalizable to state that h_α is not α-decidable, since for the program q produced by the recursion theorem in this proof, we have that q halts in less than α steps iff $f(q)$ halts in less than α steps, so the contradiction still holds, hence h_α is not α-decidable. Similarly, if H_α is α-decidable, we can make sure that the program q in the first part of Theorem 5.1 halts in less α steps, if it halts at all, so the contradiction still holds. Hence H_α is not α-decidable either. $\dashv$

Remark 5.6: The program q in the proof above needs to check that the first input p really is a natural number, and this we know from Theorem 2.5, takes ω many steps. One could imagine that this could potentially mess up this algorithm when $\alpha < \omega^2$. But we can in fact do the first ω steps in the algorithm described above and check such that $p \in \omega$ at the same time, in ω many steps. First we read p as if it was a natural number, that is, we search for the first 0 in p. When we find it (if we do not find a 0, we halt with output 'no' after ω steps), we 'assume' that the rest of the first input-tape is filled with zeros. Then we check (with the help of a bunch of flags) that this is the case at the same time as we run the algorithm above with the potentially correct natural number. If we after ω many steps discover that the rest of the first input-tape was not only 0's, we halt with output 'no', and if it was, we are all good and continues the algorithm above.

Definition 5.7a: A *snapshot* of a computation $\varphi_p(x)$ at stage α is a complete description of the machine that computes $\varphi_p(x)$ after α steps of computation, that is, a complete description of the three tapes at stage α, and a description of where the head is, and which state the program is in, at stage α. So a snapshot can be represented by an element in $2^\omega \times 2^\omega \times 2^\omega \times \omega \times \omega$ (three reals to describe the tapes, two natural numbers to describe the location of the head and the current state), so it is easy to code snapshots with reals.

Definition 5.7b: A *settled snapshot sequence* for a computation $\varphi_p(x)$ is a sequence of snapshots such that the α'th snapshot in the sequence is a snapshot of $\varphi_p(x)$ at stage α, and furthermore, the last snapshot in the sequence is either in a halting stage or the first time that the computation repeat itself in the sense of Remark 2.2. Hence, every computation $\varphi_p(x)$ has exactly one settled snapshot sequence, and it can be coded by a real, since it is a countable sequence of reals.

Theorem 5.8 (No Uniformization Theorem): There is a decidable set $A \subseteq 2^\omega \times 2^\omega$ such that for every $y \in 2^\omega$ there exists a $z \in 2^\omega$ such that $(y,z) \in A$ but such that A does not contain the graph of any computable total function.

Proof: Consider $A = \{(\langle p, x\rangle, z);\ z \text{ codes a settled snapshot sequence for } \varphi_p(x)\}$, where we choose the pairing function such that for every real y there is a real x and a natural number p such that $y = \langle p, x\rangle$.

A is decidable since we on input $(\langle p, x\rangle, z)$ can simulate $\varphi_p(x)$ and check that the snapshot sequence that z codes corresponds to the simulated computation. If at some point we find a snapshot that does not correspond to the simulated computation we answer 'no'. If this never happens one of two things can happen. Either the simulated computation halts, (and then we check if the corresponding snapshot is the last snapshot in the sequence), or we 'run out' of snapshots, (and then we check so that the last snapshot is the first snapshot where the computation repeats itself). Either way, it is decidable whether z codes a settled snapshot sequence for $\varphi_p(x)$ or not.

For every $y \in 2^\omega$ there is a $x \in 2^\omega$ and a $p \in \omega$ such that $y = \langle p, x\rangle$ and since every computation has a settled snapshot sequence, there is a $z \in 2^\omega$ such that z codes a settled snapshot sequence for $\varphi_p(x)$, and subsequently $(y, z) \in A$.

Assume now for a contradiction that A contains the graph of a total computable function f. We will specify a program that, with the help of f, decides H. On an input (p, x) the program simply simulates $f(\langle p, x\rangle)$ and decodes the result z into the settled snapshot sequence that z codes and check if the last snapshot is in a halt-stage (halt with output 'yes') or in a repeat-stage (halt with output 'no'). ⊣

Corollary 5.9: There is a program p and a real x such that the settled snapshot sequence for $\varphi_p(x)$ is not even accidentally x-writable[6].

Proof: Consider the program that on input $\langle p, x\rangle$ starts to simulate all computations on the form $\varphi_q(x)$, and after every step of computation of a simulated $\varphi_q(x)$ it checks if any of the reals currently on q's simulated tapes are a settled snapshot sequence for $\varphi_p(x)$ (this is decidable according to the No Uniformization Theorem). If it finds such a real, it is written to the output-tape and the program halts the machine. This defines a computable function f whose graph is contained in A, hence, f must not be total, because then f contradicts the No Uniformization Theorem. Hence, for at least one input $\langle p, x\rangle$ the settled snapshot sequence will not be accidentally written by a computation with input x. ⊣

Corollary 5.10: There is a total function, with a decidable graph, that is not computable.

Proof: Let f be a function such that, for every input $\langle p, x\rangle$, $f(\langle p, x\rangle) = z$, where z is the settled snapshot sequence for $\varphi_p(x)$. Then the set A in (the proof of) the No Uniformization Theorem is the graph of f. Hence, f is a total function with a decidable graph, but f can not be computable since then A would contain the graph of a computable total function, which contradict the No Uniformization Theorem. (Another way of seeing why f is not computable is to use it to solve the halting problem.) ⊣

Theorem 5.11 (Lost Melody Theorem): There is a real c such that $\{c\}$ is decidable but $c \notin$ WrR.

[6]A real is x-writable if it is writable by a machine with x as a real-oracle (to be defined in section 6) or equivalent, if it is writable with x as input (instead of 0).

Proof: For every $p \in \omega$, let z_p be a real coding the settled snapshot sequence for $\varphi_p(0)$. Let c be a real coding all the z_p's by 'slicing' the real in ω slices and have z_n on the n'th slice. Assume for a contradiction that $c \in$ WrR. We specify a program that decides h. On input p, it simply writes c to the scratch-tape and decodes the real on the p'th slice (that is z_p) into the settled snapshot sequence it codes and checks if the last snapshot is in a halt-stage (output 'yes') or in a repeat-stage (output 'no'). This program decides h which is a contradiction, hence $c \notin$ WrR.

For every $k \in \omega$ there is a φ_{p_k} that decides if the k'th slice of its input is z_k by first decoding the k'th slice of the input into (what could be) a settled snapshot sequence and then simulating $\varphi_k(0)$ and check that the simulated computation correspond to the snapshot sequence (similar to what was done in the proof of the No Uniformization Theorem). Let a program q simulate all these φ_{p_k}'s and halt with output 'no' if one of them halts with output 'no' and halt with output 'yes' if all of them has halted with output 'yes'. Then φ_q decides $\{c\}$. ⊣

COROLLARY 5.12: There is a constant total function, with a decidable graph, that is not computable.

Proof: Let f be a function such that, for every input x, $f(x) = c$, where c is the real from the Lost Melody Theorem. Then f is a constant total function, with a decidable graph. Assume that f is computable, that is, there is a $p \in \omega$ such that $\varphi_p = f$, then $\varphi_p(0) = c$ which contradicts that $c \notin$ WrR. ⊣

Our next theorem implies that the real c in the Lost Melody Theorem is not even accidentally writable.

THEOREM 5.13: Every decidable set of reals that does not contain any writable real, does not contain any accidentally writable real.

Proof: Let A be a set of reals such that $A \cap \text{WrR} = \varnothing$ but $A \cap \text{AWrR} \neq \varnothing$. We show that A is not decidable. Assume for a contradiction that A is decidable and consider a program q that simulates all computations on the form $\varphi_p(0)$, and after every step of computation in every simulated program q checks if any of the reals written to the simulated tapes in the simulated computation is in A. Since $A \cap \text{AWrR} \neq \varnothing$, such a real will eventually be found and then q copies this real to the output-tape and halts. Thus, $\varphi_q(0)$ will halt with a real from A on the output-tape which is a contradiction since $A \cap \text{WrR} = \varnothing$. ⊣

6. Oracles

There are two different kinds of oracles for IT-machines, oracles that are single reals, and oracles that are sets of reals. A machine with a real-oracle has an extra 'oracle-tape' where the real that is its oracle is written, just like an extra input. A machine with a set-oracle has an extra 'query-tape' where it can write down a real and (in one step of computation) get an answer if this real is in the oracle-set or not.

It is generally not equivalent to have x as a real-oracle or have $\{x\}$ as a set-oracle. To see this, assume that x is not even accidentally writable. A machine with x as

a real-oracle can easily halt with x on the output-tape on input 0 (just consider the program that copies the oracle-tape to the output-tape and halts). But this can not a machine with $\{x\}$ as a set-oracle do. To see why, notice that, since $x \notin$ AWrR, no real can be written to the query-tape in a computation on input 0, that would generate a positive reply from the oracle. Hence, if a machine with $\{x\}$ as an oracle halts with x on the output-tape on input 0, then there is a machine with no oracle that halts with x on the output-tape on input 0, that is, $x \in$ WrR, but that is a contradiction.

We let φ_p^x be the function that is determined by running the program p with real-oracle x, and we let φ_p^A be the function determined by running the program p with set-oracle A.

For every real x, let $M_x = \{n \in \omega;$ the n'th digit in x is $1\}$. It is easy to see that with x as a real-oracle, it is a simple task to decide if a real is in M_x or not. And with M_x as a set-oracle, it is easy to write down x and use it as a real-oracle. Hence, it is equivalent to have x as a real-oracle or M_x as a set-oracle. So in a sense, we could view the real-oracles as special cases of set-oracles, namely those consisting only of (reals coding) natural numbers.

Let A and B be oracles, then $A \leq_\infty B$ (A is computable from B) if the characteristic function of A is computable with B as an oracle. If we think of reals as subsets of ω, this definition makes sense for all kinds of oracles. As usual we let $A \equiv_\infty B$ if $A \leq_\infty B$ and $B \leq_\infty A$ and it is easy to see that this is an equivalence relation.

The notions of computability, decidability, clockability, writability and so forth are extended to A-computability, A-decidability, and so forth, in the natural way. For example, a function f is A-computable if $f = \varphi_p^A$ for some $p \in \omega$ and an ordinal α is A-clockable if there is a program that uses A as an oracle that halts after α steps of computation on input 0. Notice that $A \leq_\infty B$ iff A is B-decidable. Notice also that for every real x and every oracle B we have that $x \leq_\infty B$ iff $M_x \leq_\infty B$ iff M_x is B-decidable iff x is B-writable.

Now we define two jump operators, the weak jump and the strong jump, corresponding to the two halting problems.

Let $A \oplus B$ be the set of reals from A and B where we have added a 0 in front of all reals in A and 1 in front of all reals in B. Then both A and B are $A \oplus B$-decidable, and if A and B are C-decidable for some set of reals C, then $A \oplus B$ is also C-decidable. Hence $A \oplus B$ is in the $\leq_\infty$-smallest equivalence class that decides both A and B.

Definition 6.1a: For any oracle A, the strong jump of A, denoted $A^{\blacktriangledown}$ is the set $\{(p, x);\ \varphi_p^A(x)\downarrow\}$.
Definition 6.1b: For any oracle A, the weak jump of A, denoted $A^{\triangledown}$ is the set $A \oplus \{p;\ \varphi_p^A(0)\downarrow\}$.

As we shall see in the next theorem there are sets of reals that are not decidable from any real, so the reason for why we include A in the definition of $A^{\triangledown}$ is to make sure that $A \leq_\infty A^{\triangledown}$.

Theorem 6.2: H is not computable from any real.

Proof: Assume for a contradiction that $H \leq_\infty z$ for some real z, that is, there is a program q such that $\varphi_q^z(p,x) = 1$ if $\varphi_p(x)\downarrow$ and $\varphi_q^z(p,x) = 0$ otherwise. Let f be the computable function (defined only on ω) that on an input r, computes $s(q,r)$ (where s is such that $\varphi_q^x(r,x) = \varphi_{s(q,r)}^x(x)$), decodes $s(q,r)$ into the tuples describing the program and changes these tuples so that instead of halting, $s(q,r)$ first checks if the output is 0, and if it is, it halts, otherwise it enters a never ending loop. Then $f(r)$ is the program that halts on input x iff $\varphi_{s(q,r)}^x(x) = \varphi_q^x(r,x) = 0$. But since f is computable and total, there is a program r such that $\varphi_r = \varphi_{f(r)}$. But then we have $\varphi_r(z)\downarrow$ iff $\varphi_{f(r)}(z)\downarrow$ iff $\varphi_{s(q,r)}^z(z) = 0$ iff $\varphi_q^z(r,z) = 0$ iff $\varphi_r(z)\uparrow$. That is a contradiction. $\dashv$

This also shows, as we promised earlier, that $H \not\equiv_\infty h$, since every set of natural numbers is equivalent to some real.

THEOREM 6.3: WrR is not decidable.

Proof: Assume WrR is decidable. Consider the universal machine that simulates φ_p for all $p \in \omega$ and every time a real x is written to any of φ_p's simulated tapes, it checks if $x \in$ WrR. If it finds an x such that $x \notin$ WrR, it copies x to the output-tape and halts. Since a non-writable real eventually will be written to one of φ_p's tapes, this machine halts with a non-writable real on the output-tape on input 0. That is a contradiction. $\dashv$

COROLLARY 6.4: $A = \{(x,y);\ x \leq_\infty y\}$ is not decidable.

Proof: Since $(x,0) \in A$ iff $x \in$ WrR, we have that if A is decidable, so is WrR. Hence, the result follows from Theorem 6.3. $\dashv$

THEOREM 6.5: There are incomparable IT-degrees in the reals.

Proof: Assume that there are no incomparable IT-degrees in the reals, which implies that the degrees are linearly ordered. Since every initial segment of this order is countable (since there are only countable many programs), there are at most $\aleph_1$ degrees, so the continuum hypothesis holds. But it can be shown that the assertion that there are incomparable degrees, is a Σ_2^1-assertion (see [1]), hence this assertion is absolute to every forcing extension due to the Shoenfield absoluteness theorem. But since there is a forcing extension where the continuum hypothesis is false, there must be incomparable degrees in this extension. Hence there are incomparable IT-degrees in the reals. $\dashv$

7. Machines with only one tape (or a small head)

When we defined the IT-machines we allowed the head to read three cells at once, one from each tape. What happens if we allow the machines to read only one cell at the time? One could argue that this would be a more natural definition since the machine can write to only one cell at the time.

We start every computation with the head at the beginning of the input-tape and in the limit-stages, the head is placed at the beginning of the scratch-tape.

We would have to add two new commands for the head to be able to move up and down between the tapes.

It is fairly easy to see that these '1-read-machines' are equally powerful as the IT-machines in the sense that every IT-computable function is computable by a 1-read-machine (and vice versa). But it might just take a finite amount of steps longer since we might have to move up and down to read from the three cells that an ordinary machine reads simultaneously (it is a finite amount of steps since the difference will disappear in the limit-stages). So the question is if the ordinals clockable by a 1-read-machines are the same as the ordinary clockable ordinals.

Let ClO_{1R} be the set of ordinals clockable by a 1-read-machine.

THEOREM 7.1 (WEAK SPEED-UP): $\alpha + n \in \mathrm{ClO}_{1R} \Rightarrow \alpha + 1 \in \mathrm{ClO}_{1R}$ for all α and all $n \in \omega$.

Proof: The proof is analog to the proof of the Speed-up Lemma for the IT-machines (Theorem 3.4) except that since we can not look at two cells at once we will be forced to move a step up or down to see the other flag. All the other extra steps we might have to make will disappear in the limit-stages. ⊣

So, by the argument above, we have that $\alpha \in \mathrm{ClO} \Rightarrow \alpha + n \in \mathrm{ClO}_{1R}$ for some $n \in \omega$, hence, by Theorem 7.1, we have that $\alpha \in \mathrm{ClO} \Rightarrow \alpha + 1 \in \mathrm{ClO}_{1R}$, and consequently, every clockable successor ordinal is 1-read-clockable.

Now we will first show that there is a gap of size exactly ω^2 in the clockable ordinals, and then show that the clockable ordinals ending gaps of limit of limits length, are in fact not 1-read-clockable, hence $\mathrm{ClO} \neq \mathrm{ClO}_{1R}$.

THEOREM 7.2: The first gap of size at least ω^2 in the clockable ordinals is of size exactly ω^2.

Proof: We know from Theorem 3.6 that there is a gap of size at least ω^2 in the clockable ordinals. We will specify a universal machine that will halt after $\alpha + \omega^2$ steps, where α is the first ordinal in the first gap of size at least ω^2. We slice up the tapes in ω parts to simulate φ_p for every program p, but leaving the first cell on every tape for flags. We will use two flags, f_1 and f_2. Every time a simulated program halts, we flash f_1, and if $f_1 = 0$ in a limit-stage, we flash f_2. We halt in the first limit-stage where $f_1 = 0$ and $f_2 = 1$.

Let $f_i(\beta)$ be the value of f_i after β steps of computation.

Since no program halts between α and $\alpha + \omega^2$, we have that $f_1(\beta) = 0$ for all β such that $\alpha + \omega \leq \beta < \alpha + \omega^2$. Thus f_1 has truly stabilized to 0 before the limit-stage $\alpha + \omega^2$, hence $f_1(\alpha + \omega^2) = 0$. Also $f_1(\alpha + \omega) = 0$, $f_1(\alpha + \omega \cdot 2) = 0$,..., hence f_2 changes value unboundedly often before stage $\alpha + \omega^2$, making $f_2(\alpha + \omega^2) = 1$. Therefore, as long as this machine does not halt at some stage $\gamma \leq \alpha$, this machine halts at $\alpha + \omega^2$ and proves the theorem.

Let γ be any limit ordinal less than, or equal to α. We show that $f_2(\gamma) = 1 \Rightarrow f_1(\gamma) = 1$ which will complete the proof.

Assume that $f_2(\gamma) = 1$. Then we may assume that γ is a limit of countably many limit-stages where $f_1 = 0$, since otherwise there is a $\beta < \gamma$ such that f_1 is 1 for all stages between β and γ, and consequently $f_1(\gamma) = 1$. Since γ is a limit

of countably many limit-stages where $f_1 = 0$, there must be unboundedly large limit-stages less than γ where $f_1 = 1$, otherwise would γ end or be inside of a gap of size greater or equal to ω^2, which contradicts the choice of α. Thus we have that $f_1(\gamma) = 1$. ⊣

It is promising that we use the machines ability to read two flags at once in this algorithm since we aim to prove that this ordinal is not 1-read-clockable.

THEOREM 7.3: Every ordinal β that ends a gap of limit of limits size in the clockable ordinals, is not 1-read-clockable.

Proof: Assume $\beta \in \mathrm{ClO}_{1R}$. Since β is a limit ordinal, there has to be a program p that halts after it has read what is on the first cell C_0 on the scratch-tape directly after the limit. It is easy to see, since β is a limit of limits, that $C_0(\beta) \neq 0$, since otherwise C_0 would have stabilized at some stage $\gamma < \beta$, but the machine would then have halted at the next limit-stage (which can not be β since that limit-stage is not a limit of limits). Hence, this machine will halt in the first limit-stage where C_0 is 1. With the same argument one realizes that C_0 can not have stabilized to 1 either, it must have changed value unboundedly often before β.

For every n, there is a program p_{k_n} that simulates p but has made room for n flags that are all set to 0 at the start of the computation. After every step of computation in the simulated program p, it checks if the cell corresponding to C_0 in p is 1, and every time it is, the first flag that are 0 is turned to 1. When all n flags are 1, the computation halts. If α_n is the stage where C_0 is 1 for the n'th time in our original program, then p_{k_n} will halt at $\alpha_n + k$ for some $k \in \omega$, so the Speed-up Lemma yields that $\alpha_n \in \mathrm{ClO}$ for all $n \in \omega$. Let $\gamma = sup\{\alpha_n;\ n \in \omega\}$. Since $\alpha_n < \alpha_{n+1}$ for all $n \in \omega$, we must have that γ is a limit ordinal. And since the α_n's are unbounded in γ and $C_0(\alpha_n) = 1$ for all $n \in \omega$, we have that $C_0(\gamma) = 1$. Thus $\beta = \gamma$ which contradicts that β ends a gap (since for every $\alpha < \beta$ there is an $\alpha_n \in \mathrm{ClO}$ such that $\alpha < \alpha_n < \beta$). ⊣

This also shows that the Speed-up Lemma does not hold for 1-read-machines since for every β that ends a gap of limit of limits size in ClO, we have that $\beta \in \mathrm{ClO}$ and $\beta + 1 \in \mathrm{ClO}_{1R}$, but $\beta \notin \mathrm{ClO}_{1R}$. Thus the 1-read-machines has the same computational power as the IT-machines but $\mathrm{ClO}_{1R} \neq \mathrm{ClO}$.

How about an IT-machine with only one tape instead of three, a 1-tape-machine? Such a machine will have its only tape filled with the input at the start of the computation and when (if) it halts, the tape will be filled with the output.

In [2] Hamkins and Seabold shows that there is a 1-tape computable way of stretching the input so that it will be written to every third cell of the tape instead. They also show that when this is done, it is easy to simulate a given IT-machine by thinking of the tape as divided in three slices, with the input written to the first, and at the end of the computation, the output written to the third (with the second slice simulating the scratch-tape). They also show that there is a easy way of compressing this output to fill up the tape like it should at the end of a 1-tape computation.

Hamkins and Seabold also shows by a diagonalization argument in [2] that there is a function that is IT-computable but not 1-tape-computable.

This is confusing at first, but the 1-tape-computable functions are not generally closed under composition. To first stretch the input, then simulate the computation, then compress the output does not have to be a 1-tape-computable algorithm, even if the three parts of the algorithm is.

The reason for this is that the machine might not know that it should halt after the compression since we have no room for flags to tell us that we are done. In fact, the real at the tape after the compression might have already been written to the tape at some earlier stage of the computation.

They also show that we only need room for one single flag, that is, a cell that is not part of the output, for 1-tape-machines to compute the same functions as the IT-machines.

So how about the '1-tape-clockable' ordinals ClO_{1T}? In [2] it is shown that every clockable successor ordinal is 1-tape-clockable, which implies that the only 1-read-clockable ordinals that perhaps are not 1-tape-clockable, are the 1-read-clockable limit ordinals. Assume $\alpha \in \mathrm{ClO}_{1R}$ is a limit ordinal. Then it is easy to see that $\alpha \in \mathrm{ClO}_{1T}$. Simulate the 1-read-computation that clocks α on a 1-tape-machine and make sure that the first cell on the simulated scratch-tape is the first cell on the tape. This 1-tape-machine will halt at stage α since the finite amount of steps more that the 1-tape-machine might have to make in every step of simulated computation will disappear in the limit-stages. Hence $\mathrm{ClO}_{1R} \subseteq \mathrm{ClO}_{1T}$. And by simply using only the scratch-tape it is easy to see that a 1-read-machine can clock every ordinal that a 1-tape-machine can. Hence $\mathrm{ClO}_{1T} \subseteq \mathrm{ClO}_{1R}$ and subsequently $\mathrm{ClO}_{1T} = \mathrm{ClO}_{1R}$.

In [2] Hamkins and Seabold shows that every clockable limit of clockable ordinals is 1-tape-clockable. Which implies that the only clockable ordinals that might not be 1-tape-clockable are the ones that end gaps in the clockable ordinals. We know that every ordinal that ends a gap of limit of limits length in ClO is not 1-tape-clockable. So the only clockable ordinals that we do not know whether they are 1-tape-clockable or not are the ones that end gaps of size $\beta + \omega$ for some $\beta < \lambda$.

It is an open question whether all the ordinals that ends gaps in ClO of length $\beta + \omega$ where $\beta < \lambda$ is 1-tape-clockable, however Hamkins and Seabold shows that many of these gaps is 1-tape-clockable in the following theorem. The proof is left out because we will prove a more general theorem later.

THEOREM 7.4: For every $\delta, \beta \in \mathrm{ClO}$, the ordinal that ends the first gap after δ of size at least $\beta + \omega$ is 1-tape-clockable.

For more details on this, see [2].

DEFINITION 7.5A: Any gap of size β is a Γ_0-gap of size β.

DEFINITION 7.5B: A Γ_{n+1}-gap of size β is a gap of size β that begins with an ordinal that is a limit of ordinals that begins Γ_n-gaps of size at least β.

It is easy to see that the Γ_n-gaps with $n \neq 0$ are not covered in 7.4 since there is no clockable ordinal δ such that a Γ_n-gap (with $n \neq 0$) of size $\beta + \omega$ is the first

gap of size at least $\beta+\omega$ after δ, since this gap is a limit of such gaps (it is in fact, for every $\gamma \in \mathrm{ClO}$, not even the γ'th gap of size at least $\beta+\omega$ after δ).

The first question is if there is a Γ_n-gap (with $n \neq 0$) in the clockable ordinals. We will show that there is by specifying a program that would halt after the supremum of the clockable ordinals, λ, if no such gap existed.

Let $G^\beta(n)$ be the formula: 'There is a universal 1-tape-machine with a flag f_n such that $f_n = 0$ in a limit-stage iff it is in a stage that is $\beta+\omega$ steps after the start of a Γ_n-gap of size at least $\beta+\omega$.'

Let $F^\beta(n)$ be the formula: 'There is a universal 1-tape-machine with a flag $\overline{f}_{n+1}$ such that $\overline{f}_{n+1} = 1$ in a limit-stage iff it is in a stage that is a limit of Γ_n-gaps of size at least $\beta+\omega$.'

LEMMA 7.6: For every $\beta \in \mathrm{ClO}$, $G^\beta(n)$ and $F^\beta(n)$ holds for every $n \in \omega$.

Proof: We may assume that β is a successor ordinal since $(\beta+1)+\omega = \beta+(1+\omega) = \beta+\omega$. Hence, it is no restriction to assume that $\beta \in \mathrm{ClO}_{1T}$.

We will show the lemma by an inductive argument over F and G simultaneously. We will show, for an arbitrary $\beta \in \mathrm{ClO}_{1T}$, that

(I) $G^\beta(0)$ is true,
(II) if $G^\beta(n)$ is true, then $F^\beta(n)$ is true,
(III) if $F^\beta(n)$ is true, then $G^\beta(n+1)$ is true.

(I), (II) and (III) shows the lemma.

(I): We will not show this in the easiest way, since we will generalize this proof in (III). We need two flags, f_0 and $\overline{f}_0$. We will make sure that $f_0 = 0$ in a limit-stage iff we are $\beta+\omega$ steps into a gap of size at least $\beta+\omega$, and consequently proving (I). We define an universal machine with a β-clock (that is, a simulated computation that halts in β many steps) that makes ω steps of computation of every simulated computation and the β-clock, in ω many steps.

When a simulated computation halts: Set $\overline{f}_0$ to 1 and set f_0 to 1.
In a limit: If $\overline{f}_0$ is 1, then restart the β-clock (see Remark 7.7) and set $\overline{f}_0$ to 0, if f_0 is 0, set it to 1.
When the β-clock halts: set f_0 to 0.

Let β' be the limit ordinal such that $\beta = \beta' + n$ for some $n \in \omega$. Then it is easy to see that when the β-clock halts, we are in a gap of size at least β' (since when a simulated computation halts, we reset the β-clock in the next limit). This is the only time that f_0 is set to 0, so the only chance for f_0 to be 0 at a limit-stage is if no computation halts before the next limit. But this happens if and only if we are $\beta'+\omega = \beta+\omega$ steps into a gap of size at least $\beta+\omega$ which is exactly what we want.

(II): Consider the machine promised to exist by $G^\beta(n)$. Then we know that $f_n = 0$ in a limit-stage exactly one time in every Γ_n-gap of size at least $\beta+\omega$ and never at any other limit-stage. So flash a new flag $\overline{f}_{n+1}$ in every limit-stage where

$f_n = 0$. When $\overline{f}_{n+1} = 1$ in a limit-stage, set it to 0. Then $\overline{f}_{n+1}$ is 1 in a limit-stage iff it is a limit of Γ_n-gaps of size at least $\beta + \omega$.

(III): Consider the machine promised to exist by $F^\beta(n)$. This machine has a flag $\overline{f}_n$ that is 1 in a limit-stage iff this limit-stage is a limit of Γ_n-gaps of size at least $\beta + \omega$. This implies that all gaps that starts when $\overline{f}_n$ is 1 are Γ_{n+1}-gaps, and no gap that starts when $\overline{f}_n$ is 0 is a Γ_{n+1}-gap of size at least $\beta + \omega$. So add a new flag, f_n, and let it follow the following instructions (similar to the machine in (I)):

When a simulated computation halts: Set f_n to 1.
In a limit: If $\overline{f}_n$ is 1, then re-start the β-clock (see Remark 7.7) and set $\overline{f}_n$ to 0, if f_n is 0, set it to 1.
When the β-clock halts: Set f_n to 0.

To see that these instructions gives us the machine we want is similar to (I). $\dashv$

REMARK 7.7: There could be a problem with the β-clock, due to the restarts. When we start the clock, the slice where we clock β could be full of old 'crap' from earlier clockings or limits of clockings. This could potentially mess up the clock since it is a computation on input 0. This is however solvable.

Case 1, $\beta < \omega$: It is easy to see that it yields an equivalent machine to skip the clock and just set f_n to 0 in every limit-stage.

Case 2, $\beta \geq \omega^2$: Let p be a program that clocks β and let q be the program that in ω steps writes a 0 to every cell, regardless of how the tape was set up. Now let the real β-clock be the program that first runs q and then p. This program will halt in $\omega + \beta = \beta$ many steps on every input.

Case 3, $\omega \leq \beta < \omega^2$: There is a clockable β' such that $\omega + \beta' = \beta$. So in this case the β-clock is the program that first runs q and then runs the program that clocks β'. This program will halt in $\omega + \beta' = \beta$ many steps on every input.

Hence, for every clockable β there is a program that ends in β many steps of computation on every input.

THEOREM 7.8: For every $n \in \omega$ and every $\beta, \gamma \in \mathrm{ClO}$, there are γ many Γ_n-gaps of size at least β in ClO.

Proof: We show this with an inductive argument. We know from section 3 that, for every $\gamma, \beta \in \mathrm{ClO}$, there are γ many Γ_0-gaps of size at least β in ClO so the case $n = 0$ is already proved. Now assume that, for every $\gamma, \beta \in \mathrm{ClO}$, there are γ many Γ_n-gaps of size at least β in ClO. Then the supremum of the clockable ordinals, λ, must be a limit of Γ_n-gaps of size at least β, since for every ordinal $\mu < \lambda$, there is a clockable $\gamma > \mu$ such that there are γ many Γ_n-gaps of size at least β. Thus, at least one of these gaps must occur between μ and λ. Hence, the flag $\overline{f}_{n+1}$ in the $F^\beta(n)$-machine specified above is 1 at stage λ. If there are no Γ_{n+1}-gaps, λ would be the first limit-stage where this flag is 1 and we could easily direct the machine to halt when this happens. That is a contradiction since $\lambda \notin \mathrm{ClO}$. Hence for every $\beta \in \mathrm{ClO}$ there is a Γ_{n+1}-gap of size at least β in ClO. Now assume for a contradiction that for a certain β it is not the case that for every $\gamma \in \mathrm{ClO}$ there are γ many Γ_{n+1}-gaps of size at least β. Then there must be exactly η such gaps

for an $\eta < \lambda$. But then let σ be the supremum of the lengths of all these η gaps. Obviously $\sigma < \lambda$ so there is a $\beta' \in \mathrm{ClO}$ such that $\sigma < \beta' < \lambda$. But, we know there is a Γ_{n+1}-gap of size at least β' in ClO. This gap can not be any of the η gaps, since it is 'bigger' than every one of these. This is a contradiction since this is a gap of size at least β, and we can (finally) draw the conclusion that, for every $\beta, \gamma \in \mathrm{ClO}$, there are γ many Γ_{n+1}-gaps of size at least β, which concludes the proof. ⊣

THEOREM 7.9: If α ends the γ'th Γ_n-gap of size at least $\beta + \omega$ after δ, where $\gamma, \beta, \delta \in \mathrm{ClO}$ and $n \in \omega$, then $\alpha \in \mathrm{ClO}_{1T}$.

Proof: We may assume that δ and β are clockable successor ordinals, since the first gap after δ is the same gap as the first gap after $\delta + 1$, and since $(\beta + 1) + \omega = \beta + (1 + \omega) = \beta + \omega$. So it is no restriction to assume that β and δ is 1-tape-clockable.

Consider the 1-tape universal machine whose existence is promised by $F^{\beta}(n)$. This machine has a flag, f_n, that is 0 in a limit-stage iff this limit-stage is $\beta + \omega$ steps into a Γ_n-gap of size at least $\beta + \omega$. It will also have a flag $\overline{f}_n$ that is 1 in a limit-stage iff this limit-stage is a limit of Γ_n-gaps of size at least $\beta + \omega$. We will add a new flag $\hat{f}$ that will be set to 1 at the start of the computation and set to 0 when the next Γ_n-gap of size at least $\beta + \omega$ will be the γ'th such gap. This is how:

At the start of the computation: Make one 'IT-step' (see Remark 7.10) of computation in the simulated IT-program p that clocks γ and set $\hat{f}$ to 1.

When p halts: Set $\hat{f}$ to 0.

In a limit: If $\overline{f}_n = 1$ or if $\hat{f} = 1$ and $f_n = 0$ we make another 'IT-step' of computation in the simulation of p.

The first limit-stage where $\hat{f} = f_n = 0$ is α, since this is $\beta + \omega$ steps into the γ'th Γ_n-gap of size at least $\beta + \omega$ and we could easily direct an IT-machine to halt in the first limit-stage when $\hat{f} = f_n = 0$ since it can read three cells at once. But it is an easy thing to modify this into a 1-tape-machine by setting a master flag f to $\max(\hat{f}, f_n)$ every time we change the value of $\hat{f}$ or f_n. Then $f = 0$ in a limit-stage iff $\hat{f} = f_n = 0$ in this limit-stage. So by placing this master flag on the first cell of the tape, we can easily halt at stage α and the proof is complete. ⊣

REMARK 7.10: If an IT-program p clocks an $\alpha \in \mathrm{ClO}$, we can make this into a 1-tape-computation simply by slicing the tape into three, and simulate the three tapes of this IT-machine on these slices. Since the input is 0, there is no need to stretch it, and we do not care about the output in this particular case so there is no need for compressing the output either. If we by an 'IT-step' of computation mean the finite amount of steps this 1-tape-machine does to simulate one step of computation in the original IT-machine (it has to move around and read from three cells before it consults the IT-program p). Then it is easy to see that this 1-tape-computation will halt after α 'IT-steps' of computation.

The proof of Theorem 7.9 shows that for every $\beta, \gamma, \delta \in \mathrm{ClO}$, the γ'th Γ_n-gap of size at least $\beta + \omega$ after δ is of size exactly $\beta + \omega$. Hence every Γ_n-gap of size greater than $\beta + \omega$ is the $\overline{\gamma}$'th gap of size at least $\beta + \omega$, after δ, for some $\overline{\gamma} \notin \mathrm{ClO}$.

There are (at least) two things that might cause Theorem 7.9 to fail to cover all the gaps we want it to.

1: The theorem does not cover any gaps of non-clockable length. Thus, one needs to show that no such gaps exists or one needs to strengthen the theorem.

2: There could be Γ_α-gaps for $\omega \leq \alpha < \lambda$. If so, one needs to generalize the theorem to a more general induction.

8. Infinite time machines with infinite programs (ISIT-machines)

If we in the definition of the IT-machines let go of the restriction that the programs must be finite sets and instead let the programs be countable sets, we get the ISIT-machines. If we dropped the restriction that the ISIT-programs must be countable, it would not yield a more powerful machine (see Remark 8.7).

Thus, a program is a countable set of tuples $\langle s, r, c, s' \rangle$ that specifies the machines actions. It is obvious that the IT-machines are those ISIT-machines with a finite program.

It is no restriction to let the set of all possible states be ω since there can only be countably many states per program. So the set of possible tuples are exactly the same as for the IT-programs. Hence there is a recursive bijection between the set of possible tuples and ω. So every program is coded by a subset of ω (and every subset of ω codes a program). And since every real codes a subset of ω in an obvious way we will be careless, as usual, and call a real x a program when we really mean the program that x codes.

And, like before, every program x determines a partial function φ_x by letting the value of $\varphi_x(z)$ be what is on the output-tape when the machine with program x and input z halts, and letting $\varphi_x(z)$ be undefined when the machine does not halt. Of course $\varphi_x = \varphi_y$ iff $\varphi_x(z)\downarrow \leftrightarrow \varphi_y(z)\downarrow$ for all z and $\varphi_x(z) = \varphi_y(z)$ for all z such that $\varphi_x(z)\downarrow$.

We extend, in the natural way, the notions of computability, (semi-)decidablilty, writability, and so forth, to the ISIT-context. For instance, a function f on reals is ISIT-computable if there is a program x such that $f = \varphi_x$.

THEOREM 8.1: Every real x is ISIT-writable in ω steps of computation.

Proof: Fix an $x \in 2^\omega$. We show that there is a machine that writes x on the output-tape on input 0. Consider this program:
$\{\langle 2n+1, r, \text{write the } n\text{'th digit of } x \text{ on the output-tape}, 2n+2\rangle\,;\ n \in \omega, r \in R\} \cup$
$\{\langle 2n+2, r, \text{move one step to the right}, 2n+3\rangle\,;\ n \in \omega, r \in R\}$ (where R is the set of possible reads). This machine writes x on the output-tape in ω steps, and in the limit-stage it halts since there is no state 0 in the program. $\dashv$

Hence 'writable' is a meaningless concept for ISIT-machines. We also see that the ISIT-machines are 'stronger' than the IT-machines, since Theorem 8.1 shows that there are uncountably many ISIT-computable functions.

COROLLARY 8.2: There are $2^{\aleph_0}$ different ISIT-computable functions.

Proof: Since every program is coded by a real, there must be at most $2^{\aleph_0}$ different ISIT-computable functions, but Theorem I implies that there are at least $2^{\aleph_0}$ different ISIT-computable functions, so there must be exactly $2^{\aleph_0}$ different ISIT-computable functions. ⊣

THEOREM 8.3: For every ISIT-computable function f there are $2^{\aleph_0}$ different indices x such that $f = \varphi_x$.

Proof: Let x be a program. Multiply every state in x (except 0 and 1) with 2. This gives us a new program x' such that $\varphi_x = \varphi_{x'}$ with ω many free states. So we can add pointless states to this program in 2^ω different ways. All these programs produce different indices but defines the same function. ⊣

THEOREM 8.4 (S-N-M): There is an ISIT-computable, total function s such that $\varphi_x(\overline{y}, \overline{z}) = \varphi_{s(x,\overline{y})}(\overline{z})$

Proof: Assume we have a program x with $n + m$ arguments. We will specify the program $x' = s(x, \overline{y})$ with m arguments. First it slices up the scratch-tape in $n + 1$ slices and writes the $\overline{y}$'s on n of them, leaving one for scratch work. Then it runs the same algorithm as x except that it reads the $\overline{y}$'s from slices of the scratch-tape instead of input-tapes, and it does scratch work on a slice of the scratch-tape instead of the whole scratch-tape. This procedure is ISIT-computable. So for an ISIT-program that gets x and $\overline{y}$ as input it is a easy thing to decode x into tuples, make these changes, and code it back into $x' = s(x, \overline{y})$. Hence, s is ISIT-computable. ⊣

THEOREM 8.5 (RECURSION): For every total ISIT-computable function f there is a program x such that $\varphi_x = \varphi_{f(x)}$.

Proof: The classical proof works in this context as well. Fix f. Let μ be the program so that $\varphi_\mu(y, z) = \varphi_{f(s(y,y))}(z)$. Let $x = s(\mu, \mu)$. Then we have $\varphi_x(z) = \varphi_{s(\mu,\mu)}(z) = \varphi_\mu(\mu, z) = \varphi_{f(s(\mu,\mu))}(z) = \varphi_{f(x)}(z)$. ⊣

THEOREM 8.6: Every halting ISIT-computation is countable.

Proof: It is not hard to see that Remark 2.2 is true for ISIT-machines as well, so the proof of the corresponding theorem for IT-machines (Theorem 2.3) works in this context. ⊣

REMARK 8.7: If we allowed the programs to be any set of tuples without the restriction that it must be countable, and let any ordinal be a possible state (instead of letting ω be the set of possible states), it would not yield a more powerful machine. Remark 2.2 and Theorem 2.3 would be true in this context as well, making every halting computation countable and every non-halting computation caught in a countable loop. Thus, only a countable subset of the program is ever used, the rest is just dummy-tuples, and subsequently, only countably many states is used. Hence, every such machine can be simulated by an ISIT-machine.

THEOREM 8.8: Every ordinal $\alpha < \omega_1$ is clockable.

Proof: Just like for IT-machines, it is easy to see that all $\alpha < \omega^2$ is clockable (see section 3). Assume $\alpha \geq \omega^2$. According to Theorem 5.2 there is an IT-machine p that halts after α steps of computation on some input $x \in 2^\omega$. Construct a ISIT-program that on input 0 first writes x on the input-tape and then runs the exact same algorithm as p. This machine will halt after $\omega + \alpha = \alpha$ steps on input 0. $\dashv$

Theorem 8.6 and Theorem 8.8 tell us that all ordinals that can be clockable is, hence 'clockable', just like 'writable', is not a meaningful concept for ISIT-machines. It is also easy to see that the set of writable ordinals is the same as the set of clockable ordinals.

Let $A \,\widehat{\leq}_\infty\, B$ if the characteristic function of A is ISIT-computable with B as an oracle, and let $A \,\widehat{\equiv}_\infty\, B$ if $A \,\widehat{\leq}_\infty\, B$ and $B \,\widehat{\leq}_\infty\, A$. Of course, since every real is ISIT-writable, only set-oracles are meaningful in the ISIT context.

Let $\widehat{H}$ and $\widehat{h}$ be the halting problems relativized to ISIT-machines, that is, $\widehat{H} = \{(x, y);\ \varphi_x(y)\downarrow\}$ and $\widehat{h} = \{x;\ \varphi_x(0)\downarrow\}$. It is easy to see that $\widehat{H}$ and $\widehat{h}$ are semi-decidable.

THEOREM 8.9: $\widehat{H} \,\widehat{\equiv}_\infty\, \widehat{h}$.

Proof: Obviously $\widehat{h} \,\widehat{\leq}_\infty\, \widehat{H}$. We show that $\widehat{H} \,\widehat{\leq}_\infty\, \widehat{h}$. Suppose we want to decide if $(x, y) \in \widehat{H}$, that is if $\varphi_x(y)\downarrow$. Since y is writable, there is a program $f(x, y)$ such that f is computable and the program $f(x, y)$, on input 0, first writes y on a slice of the scratch-tape and then runs (practically) the same algorithm as x. Then we have that $\varphi_x(y)\downarrow$ iff $\varphi_{f(x,y)}(0)\downarrow$ and this we can decide with $\widehat{h}$ as an oracle. $\dashv$

THEOREM 8.10: $\widehat{H}$ and $\widehat{h}$ are not ISIT-decidable.

Proof: The corresponding proof of that H is not IT-decidable (Theorem 5.1) works in this context to show that $\widehat{H}$ is not ISIT-decidable. Thus, Theorem 8.9 implies that $\widehat{h}$ is not ISIT-decidable. (The corresponding proof of that h is not IT-decidable does not work in this context since it uses that h is countable). $\dashv$

Let $\widehat{H}_\alpha = \{(x, y)$; the program x halts in less than α steps of computation on input $y\}$ and $\widehat{h}_\alpha = \{(x, y)$; the program x halts in less than α steps of computation on input $0\}$. Then we have:

(1) $\widehat{H}_\alpha \subsetneq \widehat{H}_\beta$, for all $\alpha < \beta \leq \omega_1$
(2) $\widehat{H}_{\omega_1} = \widehat{H}_\alpha = \widehat{H}$, for all $\alpha \geq \omega_1$
(3) $\widehat{H}_\alpha$ is ISIT-α-semi-decidable but not ISIT-α-decidable, for all limit α
(4) $\widehat{H}_\alpha$ is ISIT-$(\alpha + 1)$-decidable, for all limit α
(5) $\widehat{H}_\alpha$ is ISIT-decidable for all $\alpha < \omega_1$.

(1) follows from Theorem 8.8, (2) follows from Theorem 8.6 and (3)-(5) has basically the same proof as corresponding IT-statements (see section 5). (1)-(5) also holds for $\widehat{h}$.

THEOREM 8.11: There is an IT-program q such that for every ISIT-program x we have $\varphi_x = \varphi_q^x$.

Proof: We specify the IT-program q, that with x as a real-oracle, will simulate the ISIT-program x.

At 'time' v (the 'time' v here will correspond to the state the simulated computation currently is in):

The first time q does this, it does it with $v = 1$ and after that q knows what v is from the previous round of computation. The first thing it does is 'memorize' what it reads r from the tape and then it starts decoding x (which it has access to since x is its oracle) into the tuples describing x's program. If it finds a tuple $\langle v, r, c, w\rangle$ (where v is the current 'time'), it moves back to where it read r and executes command c, flashes a flag on and of, and starts over (unless c tells it to halt) at 'time' w. This will be done in a finite amount of steps (as long as it finds the corresponding tuple).

In limit-stages it checks if the flag is on or not. If it is on it will start over at 'time' 0, if it is off, that means that at some point we did not find the corresponding tuple and searched for it in x in vain, so then we halt. This IT-program q will, on every input $\overline{y}$, halt with $\varphi_x(\overline{y})$ as output if $\varphi_x(\overline{y})$ halts, and run forever if $\varphi_x(\overline{y})$ is undefined, so $\varphi_x = \varphi_q^x$. ⊣

From now on q will be the IT-program in 8.11 that simulates the ISIT-program that it has as a real-oracle.

COROLLARY 8.12: The power of ISIT-machines are exactly the same as the power of IT-machines with real-oracles.

Proof: Theorem 8.11 shows that the ISIT-machines are not stronger than IT-machines with real-oracles. To see that IT-machines with real-oracles are not stronger than the ISIT-machines, consider for every IT-program p with x as an oracle, the ISIT-program that first writes x on a slice of the scratch-tape and then runs basically the same algorithm as p. ⊣

The proof of 8.11 uses the easy coding we have chosen to go from a real x to the program x codes, and back. One could imagine that this proof might not go through with an ISIT-computable coding-algorithm that is not IT-computable, but Corollary 8.12 still holds since the strength of the machines does not depend on the coding.

COROLLARY 8.13: For every ISIT-computable function φ_x where x is IT-writable there is an IT-computable function φ_p such that $\varphi_x = \varphi_p$.

Proof: First we write x on a slice of the scratch-tape and then we follow practically the same algorithm as in the proof of Theorem 8.11. ⊣

This implies for example that the recursive ISIT-machines are not stronger than the IT-machines.

Let $H = \{(x,y);\ \varphi_x(y)\downarrow \wedge\ x \text{ codes a finite program}\}$ and
$h = \{x;\ \varphi_x(0)\downarrow \wedge\ x \text{ codes a finite program}\}$. So H and h are (equivalent to) the halting problems for the IT-machines.

COROLLARY 8.14: H is not ISIT-decidable.

Proof: Assume φ_x decides H. Theorem 8.11 implies that φ_q^x decides H. But that contradicts Theorem 6.2. ⊣

THEOREM 8.15: $\widehat{H} \mathrel{\widehat{\equiv}}_\infty H$.

Proof: Since it is easy to check if x codes a finite program, it is clear that $H \mathrel{\widehat{\leq}}_\infty \widehat{H}$. Now we show that $\widehat{H} \mathrel{\widehat{\leq}}_\infty H$. Let q' be the program such that $\varphi_{q'}(x,y) = \varphi_q^x(y)$. This program exists since it is easy to direct q to look at the first input-tape instead of the oracle-tape. Then let q'' be such that $\varphi_{q'}(x,y) = \varphi_{q''}(\langle x,y\rangle)$. But then we have that $(x,y) \in \widehat{H}$ iff $\varphi_x(y)\downarrow$ iff $\varphi_q^x(y)\downarrow$ iff $\varphi_{q'}(x,y)\downarrow$ iff $\varphi_{q''}(\langle x,y\rangle)\downarrow$ iff $(q'', \langle x,y\rangle) \in H$. ⊣

Since $\widehat{H} = \{(x,y);\ \varphi_q^x(y)\downarrow\}$, we see that the proof above also shows that $\widehat{H} \equiv_\infty H$ (where $\equiv_\infty$ is the relation defined in section 6).

THEOREM 8.16: The relation $x \leq_\infty y$ is not ISIT-decidable (where $\leq_\infty$ is the relation defined in section 6).

Proof: Assume there is an ISIT-program z such that φ_z decides $A = \{(x,y);\ x \leq_\infty y\}$. Then we have that φ_q^z decides A. But then there is an IT-program $\overline{q}$ such that $\varphi_{\overline{q}}^z(x) = \varphi_q^z(x,z)$. This implies that $B = \{x;\ x \leq_\infty z\} = \{x;\ x \text{ is } z\text{-writable}\}$ is IT-decidable with z as an oracle since $\varphi_{\overline{q}}^z$ decides B. This is a contradiction. To see why, consider the IT-program q' that, with z as an oracle, simulates every computation on the form $\varphi_p(z)$ on a slice of the scratch-tape, and as soon as a real on any of the simulated tapes have been altered, q' checks if this real is a member of B. If it is, q' continues with the computation, but if it is not, q' copies this real to the output-tape and halts. This machine will eventually halt with a real that is not z-writable on the output-tape (to see why, generalize Theorem 3.12 to show that there are eventually z-writable reals that are not z-writable). That is a contradiction. ⊣

THEOREM 8.17: Every countable set of reals is ISIT-decidable.

Proof: For every countable set there is a program that slices up the scratch-tape in ω slices and simply writes up all the members of the set. It is then a simple task to check if a given input has been written to any of the slices. ⊣

COROLLARY 8.18: h is ISIT-decidable.

Proof: Follows directly since h is countable. Another way of seeing it is to consider the program that writes the supremum of the IT-clockable ordinals λ and then simulates the IT-machine in question on input 0 and at the same time counts down λ. The result also follows from the fact that c from the Lost Melody Theorem is ISIT-writable. ⊣

COROLLARY 8.19: ClO, WrO, EWrO, AWrO, WrR, EWrR and AWrR are all ISIT-decidable.

Proof: Since ClO $\subseteq$ WrO $\subseteq$ EWrO $\subseteq$ AWrO and WrR $\subseteq$ EWrR $\subseteq$ AWrR and $|\text{AWrO}| \leq |\text{AWrR}|$, it suffices to show that AWrR is countable and the result follows from Theorem 8.17. But that follows from Theorem 2.3 and Corollary 2.4, since they imply that every program produces at most countably many accidentally writable reals. ⊣

THEOREM 8.20: Every ISIT-decidable and ISIT-semi-decidable set is $\underline{\Delta^1_2}$.[7]

Proof: Assume that φ_x (semi-)decides A. This means that φ^x_q (semi-)decides A. This means that A is IT-(semi-)decidable with x as a real-oracle. This means (according to section 7 in [1]) that A is $\underline{\Delta^1_2}$. ⊣

9: Conclusions

In the proof of that ClO $\subseteq$ WrO, we discover that Statement 4.4 is used (although never stated) without proof in [3]. It may very well be obvious that Statement 4.4 is true and that we just have failed to see it, otherwise, one would need to prove it to complete the proof.

We did not succeed in solving the open question in [2], that is, 'is every ordinal that ends a gap of size $\beta + \omega$ (for some $\beta < \lambda$) in the clockable ordinals 1-tape-clockable?', but at least we showed that even more of them, than was known before, are.

Thanks to Theorem 8.11, most of the questions we had about the ISIT-machines where answered with relative ease. We showed that the power of the ISIT-machines is equivalent to the power of IT-machines with real-oracles (in the sense that the same functions are computable). We also showed that it suffices for the ISIT-program to be IT-writable for the resulting function to be IT-computable, and hence it is indeed true for recursive programs. We also showed that h is ISIT-decidable but that H is equivalent to the halting problem for ISIT-machines, and therefore not ISIT-decidable.

[7] A is $\underline{\Pi^1_1}$ if there is a second-order arithmetical formula $\forall \overline{X} \psi(\overline{X}, Y, \overline{B})$, where $\overline{B}$ are set parameters and ψ is a first-order formula, that defines A in the standard model.

References

[1] Joel David Hamkins and Andy Lewis. Infinite time Turing machines. *J. Symbolic Logic*, 65(2):567–604, 2000.

[2] Joel David Hamkins and Daniel Evan Seabold. Infinite time Turing machines with only one tape. *MLQ Math. Log. Q.*, 47(2):271–287, 2001.

[3] P. D. Welch. The length of infinite time Turing machine computations. *Bull. London Math. Soc.*, 32(2):129–136, 2000.