Post's Problem for ordinal register machines: an explicit approach

Joel David Hamkins

College of Staten Island of The City University of New York
Mathematics Dept., 2800 Victory Boulevard, Staten Island, NY 10314
The Graduate Center of The City University of New York
Ph.D. Programs in Mathematics and in Computer Science
365 Fifth Avenue, New York, NY 10016, USA

Russell G. Miller

Queens College of The City University of New York
Mathematics Dept., 65-30 Kissena Boulevard, Flushing, New York 11367
The Graduate Center of The City University of New York
Ph.D. Programs in Computer Science and in Mathematics
365 Fifth Avenue, New York, NY 10016, USA

Abstract

We provide a positive solution for Post's Problem for ordinal register machines, and also prove that these machines and ordinal Turing machines compute precisely the same partial functions on ordinals. To do so, we construct ordinal register machine programs which compute the necessary functions. In addition, we show that any set of ordinals solving Post's Problem must be unbounded in the writable ordinals.

Key words: Computability, ordinal computability, ordinal register machine, Post's Problem.

⋆ Much of this paper was presented by the second author at the Computability in Europe meeting, Siena, Italy, 19 June 2007, and appeared in an extended abstract in [4].

[1] The research of the two authors has been supported in part by grants from the Research Foundation of CUNY and by support for their attendance at the Bonn International Workshop in Ordinal Computability. The first author is additionally thankful to the Institute for Logic, Language and Computation and the NWO (Bezoekersbeurs B62-612) for supporting his stays at the Universiteit van Amsterdam in 2006 and 2007.

1 Definitions

Ordinal register machines, or ORM's, are defined and described by Koepke and Siders in [8]. They generalize the traditional finite-time register machines: the registers are now allowed to contain any ordinal value, not just natural numbers, and the (finite) program runs through ordinal time, with its state at limit-ordinal stages determined in a natural way by taking liminf's of the cells and states at the preceding stages.

Koepke and Siders proved in [8] that the sets of ordinals computable by an ORM, with finitely many ordinal parameters, are precisely the constructible sets of ordinals, i.e. those lying in Gödel's constructible universe L. Various other results on computability in ordinal time and space have been proven by set-theoretic methods. (See [2,6,7], for example.) These proofs tend to be quick and clean, and avoid actually dealing with the ORM's themselves. Our intention in this paper is to develop actual ORM programs to answer two of the basic questions about computability under such machines. We will address Post's Problem for ORM's, and also compare the computational capabilities of ORM's with those of the ordinal Turing machines (OTM's) described by Koepke in [6]. In doing so, we will illustrate the power of the special type of ORM known as an *ordinal stack machine*, first described in [8].

Since the ordinal register machine programs are finite, they each refer to only finitely many registers, and the memory used by any ORM algorithm is correspondingly limited to these fixed finite number of ordinal values at any time. This contrasts with the situation for OTM's, and also with the situation for the infinite time Turing machines of Hamkins and Lewis [2], where the algorithms can store information stretching out on a transfinite tape. One is led to suspect that ORM's are less powerful than OTM's, but in fact we will show that they compute precisely the same class of partial functions on ordinals.

The original version of Post's Problem applied to finite-time Turing machines. It asked whether there exists a computably enumerable set A which is neither computable nor complete. That is, it required $\emptyset <_T A <_T \emptyset'$, where $\emptyset'$ is the jump of the empty set, or equivalently the Halting Problem for finite-time Turing machines. Post's Program for solving this problem was to discover a nonvacuous property of c.e. sets, expressible using only the containment relation $\subseteq$, which would guarantee that A was incomplete and noncomputable.

Post's Program was the genesis of the notions of simple, hypersimple, and hyperhypersimple sets, all of which properties Post originally hoped would fulfill his program. In fact, none of these properties implies incompleteness, and Post did not live to see the solution of the problem that bears his name. Post's Program was completed by Harrington and Soare [5] in 1991, but his

Problem was solved much earlier, in 1956 and 1957, with the invention of the finite injury priority method (independently) by Friedberg [1] and Muchnik [9]. A good description of this method appears in section VII.2 of [12].

The notion of a computably enumerable subset of ω extends naturally to our context: a set of ordinals is *ORM-enumerable*, or *semidecidable*, if it is the domain (equivalently, the range) of some ORM-computable function. Shortly we will also define the jump operation for ORM's. Then we will ask the analogue of Post's Problem for sets of ordinals under computation by ORM's.

We often conflate ORM's with the partial functions they compute, which we enumerate as $\varphi_0, \varphi_1, \ldots$ by effectively coding their programs, just as for finite-time Turing and register machines. In this context we disregard ORM's with ordinal parameters. Functions of arity > 1 can be considered by effectively identifying finite tuples of ordinals with single ordinals.

We write $\varphi_{e,\sigma}(\alpha) \downarrow$ to signify that the program φ_e converges on input α in *strictly fewer than* σ steps. This notation is different from the common usage in finite-time computability theory, where $\varphi_{e,s}(n) \downarrow$ denotes convergence in $\leq s$ steps; our way is more appropriate in a context where we must deal with limit ordinals.

An ordinal α is ORM-*writable* if there is an ordinal register machine φ_e which, on input 0, halts and outputs α. Briefly, $\varphi_e(0) \downarrow = \alpha$ for some $e \in \omega$. Also, an ordinal σ is ORM-*clockable* if some computation $\varphi_e(0)$ halts after exactly σ steps: $\varphi_{e,\sigma}(0) \uparrow$, but $\varphi_{e,\sigma+1}(0) \downarrow$. Notice that it is ORM-computable whether σ is clockable, since we can run all computations $\varphi_e(0)$ for σ-many steps and check whether any of them halted after exactly σ steps. On the other hand, the set of writable ordinals is ORM-enumerable, but not ORM-computable, essentially because a computation which halts after a very long number of steps could still have a relatively small ordinal as its output.

(To see that φ_e does not compute the set of writable ordinals, consider a program q which, on input 0, goes through every pair $\langle \alpha, \sigma \rangle$ of ordinals in turn and runs $\varphi_e(\alpha)$ for σ steps. If it finds that $\varphi_{e,\sigma}(\alpha) \downarrow = 0$, then it copies that α into its output register and halts, thus contradicting φ_e's claim by showing that α is writable, even though $\varphi_e(\alpha) = 0$. Otherwise it goes on to the next pair $\langle \alpha, \sigma \rangle$. But if this program q never halts, then $0 \notin \text{range}(\varphi_e)$, so either way φ_e does not compute the set of writable ordinals.)

Oracle computation for ORM's, with an oracle X which is a set or class of ordinals, is normally defined by choosing one specific register r_i as the *oracle register* and allowing the machine to execute instructions of the form "if the content of r_i lies in X, then execute instruction number j." Intuitively, the oracle will answer any question about membership of individual ordinals in X. Notice that to ask whether $\gamma \in X$, however, the machine itself must first

write γ in r_i, possibly using its input to do so. We write $Y \leq_{ORM} X$ to denote that some ORM with oracle X computes the characteristic function of a set Y of ordinals.

The *weak jump* $\emptyset^\diamondsuit$ of the empty set for ORMs is the set

$$\emptyset^\diamondsuit = \{e \in \omega : \varphi_e(0)\downarrow\}.$$

We have approximations to the weak jump:

$$\emptyset^\diamondsuit_\sigma = \{e : \varphi_{e,\sigma}(0)\downarrow\};$$

these are nested upwards and are computable uniformly in σ. Also notice that σ is clockable iff $\emptyset^\diamondsuit_\sigma \neq \emptyset^\diamondsuit_{\sigma+1}$, and that for limit ordinals λ, $\emptyset^\diamondsuit_\lambda = \cup_{\sigma<\lambda}\emptyset^\diamondsuit_\sigma$. (This would fail if we had kept the finite-time notation for $\varphi_{e,\lambda}(\alpha)\downarrow$.) We also have the *strong jump* $\emptyset^\blacklozenge$ of $\emptyset$ and its computable approximations $\emptyset^\blacklozenge_\sigma$:

$$\emptyset^\blacklozenge = \{\langle e, \alpha\rangle : \varphi_e(\alpha)\downarrow\} \subset \omega \times \mathrm{ON} \qquad \emptyset^\blacklozenge_\sigma = \{\langle e, \alpha\rangle : \varphi_{e,\sigma}(\alpha)\downarrow\}.$$

The strong jump is the actual halting problem in ORM-computability; the weak jump, roughly analogous to the jump in finite-time computability, is just the most convenient way to diagonalize and build a noncomputable set. In finite time, of course, the jump and the halting problem are computably isomorphic, but in our context this is no longer true; indeed $\emptyset^\diamondsuit <_{ORM} \emptyset^\blacklozenge$, with strict inequality.

The version of the following lemma for infinite time Turing machines was a significant result, proved by Philip Welch in [13], but in the ordinal register machine context we observe it easily:

Lemma 1 *For ordinal register machines, the supremum γ of the clockable ordinals equals the supremum λ of the writable ordinals.*

PROOF. Every clockable ordinal α is writable: just run the computation $\varphi_p(0)$ which halts after α steps, adjusting the machine so that at each step, it increments the ordinal in a new step register. When $\varphi_p(0)$ halts, transfer the contents of the step register to the output register and then halt. Thus $\gamma \leq \lambda$. Conversely, if $\alpha = \varphi_q(0)$ is writable, then $\varphi_q(0)$ takes at least α many steps to halt, since after β steps, an easy induction shows that no register can contain any ordinal larger than β. Hence $\lambda \leq \gamma$. $\square$

2 Post's Problem

Now we begin to consider Post's Problem. First we ask whether there exist relatively simple ORM-enumerable sets (subsets of ω, for instance) which are noncomputable and incomplete. The answer is no.

Theorem 2 *No subset $C \subseteq \omega$ satisfies $\emptyset <_{ORM} C <_{ORM} \emptyset^{\diamondsuit}$. Indeed, the same holds for subsets $C \subseteq \rho$, for any writable ordinal ρ.*

PROOF. Consider a set $C \subseteq \rho$ with $\emptyset \leq_{ORM} C \leq_{ORM} \emptyset^{\diamondsuit}$ and ρ writable. This part of the proof is similar to that of Theorem 2.1 in [3], the corresponding result for infinite time Turing machines.

Recall that $\emptyset^{\diamondsuit}_{\sigma} = \{e \in \omega : \varphi_{e,\sigma}(0)\downarrow\}$. This is a computable enumeration of the semidecidable set $\emptyset^{\diamondsuit}$. By assumption there is a program q such that $\varphi_q^{\emptyset^{\diamondsuit}}$ computes the characteristic function of C. This gives us a computable approximation to C:

$$C_\sigma = \{\beta < \rho : (\exists \delta \geq \sigma)[\varphi_{q,\delta}^{\emptyset^{\diamondsuit}_\sigma}(\beta)\downarrow = 1 \ \&\ (\forall \theta)[\sigma < \theta \leq \delta \implies \emptyset^{\diamondsuit}_\sigma = \emptyset^{\diamondsuit}_\theta]]\}.$$

Since $\varphi_q^{\emptyset^{\diamondsuit}}$ is the characteristic function of C, it must be total. Indeed, since $\emptyset^{\diamondsuit}_\gamma = \emptyset^{\diamondsuit}$, we must have $C_\sigma = C$ for all $\sigma \geq \gamma$. Therefore we can compute, uniformly in β and σ, whether $\beta \in C_\sigma$: having written ρ and checked that $\beta < \rho$, just run $\varphi_q^{\emptyset^{\diamondsuit}_\sigma}(\beta)$ until we reach a stage $\delta \geq \sigma$ such that either $\emptyset^{\diamondsuit}_\delta \neq \emptyset^{\diamondsuit}_\sigma$ (so $\beta \notin C_\sigma$) or $\varphi_{q,\delta}^{\emptyset^{\diamondsuit}_\sigma}(\beta)\downarrow$. If we find that this computation halts and outputs 1, before the approximation to $\emptyset^{\diamondsuit}$ changes, then $\beta \in C_\sigma$; if it outputs a different value, then $\beta \notin C_\sigma$. (In finite-time computability, the analogous process is the construction of a computable approximation to an arbitrary set $\leq_T \emptyset'$.)

Lemma 3 *With this approximation, if $C_\sigma \neq C_{\sigma+1}$, then $\sigma + 1$ is clockable (and hence σ is writable).*

PROOF. If $\emptyset^{\diamondsuit}_\sigma \neq \emptyset^{\diamondsuit}_{\sigma+1}$, then some computation $\varphi_e(0)$ converged in $(\sigma+1)$-many steps, so $(\sigma+1)$ is clockable. Otherwise, the definition of C_σ shows that $C_\sigma = C_{\sigma+1}$, as follows. To check whether some $\beta < \rho$ lies in C_σ, we find the least $\delta \geq \sigma$ for which either $\emptyset^{\diamondsuit}_\sigma \neq \emptyset^{\diamondsuit}_\delta$, or $\varphi_{q,\delta}^{\emptyset^{\diamondsuit}_\sigma}(\beta)\downarrow$. If $\delta \geq \sigma + 1$, then we find the same δ when checking whether $\beta \in C_{\sigma+1}$, so the answer is the same. If $\delta = \sigma$, then $\varphi_{q,\sigma}^{\emptyset^{\diamondsuit}_\sigma}(\beta)\downarrow$, and hence $\varphi_{q,\sigma+1}^{\emptyset^{\diamondsuit}_{\sigma+1}}(\beta)\downarrow = \varphi_{q,\sigma}^{\emptyset^{\diamondsuit}_\sigma}(\beta)$ as well, since the oracle has not changed. $\square$

We now consider two cases. First, if there exists some $\beta < \gamma (= \lambda)$ such that $C_\beta = C$, then C is ORM-computable. To compute C, we run some

fixed program $\varphi_p(0)$ which writes the least writable ordinal $\delta \geq \beta$ and then computes C_δ. By Lemma 3, $C_\sigma = C_\beta = C$ for every σ with $\beta \leq \sigma < \delta$. But then $\emptyset^\Diamond_\delta = \emptyset^\Diamond_\beta$ as well, since $\emptyset^\Diamond_\delta$ contains those programs which halt *before* stage δ, and so the definition of C_σ shows that $C_\delta = C_\beta = C$. Hence we have computed C.

Otherwise there is no such β, and in this case $\emptyset^\Diamond \leq_{ORM} C$, since with a C-oracle we can search for the least σ such that $C_\sigma = C$. (Here we need to know that C is contained in ρ, as assumed by the theorem. We can write ρ, and then to verify that $C = C_\sigma$, we need only check that all $\alpha < \rho$ lie in C iff they lie in C_σ.) But by assumption, the σ we find is $\geq \gamma$ (in fact precisely γ), and when we find it, we write it on the output tape and halt. Thus γ is C-writable, and with γ it is easy to compute $\emptyset^\Diamond$. $\Box$

So Post's Problem has a negative solution when we restrict to sets C such that the supremum of C is writable. This is a large class of sets: it includes every non-cofinal subset of γ. However, without this restriction, the same problem has a positive solution. We prove this by a construction in the style of Friedberg and Muchnik. First we give a necessary lemma.

Lemma 4 (Reflection Lemma for ORM's) *Suppose $\varphi^A_e(x) \downarrow = 0$, where x is a writable ordinal and A is a semidecidable set of ordinals with ORM-computable enumeration $\langle A_\sigma \rangle$ such that $A_\gamma = A$. (This means that $A_\sigma \subseteq A_\tau$ for all $\sigma < \tau$, that $A_\beta = \cup_{\sigma<\beta} A_\sigma$ for limit ordinals β, and that there is a computable function $f(\alpha, \sigma)$ with value 1 if $\alpha \in A_\sigma$ and 0 if not.) Assume also that every σ with $A_\sigma \neq A_{\sigma+1}$ is clockable. Then for every β less than the supremum γ of the clockable ordinals, there exists a clockable limit ordinal $\tau > \beta$ such that $\varphi^{A_\tau}_{e,\tau}(x) \downarrow = 0$.*

The same would hold if we replaced "clockable" by "writable" throughout the lemma. However, clockability will be the property we need.

PROOF. Our proof mirrors that of the Reflection Lemma for infinite time Turing machines (Lemma 4.3 in [3]). Consider the algorithm which, on input 0, writes x and then searches and outputs the least nonclockable ordinal $\sigma > \beta$ such that $\varphi^{A_\sigma}_{e,\sigma}(x) \downarrow = 0$.

Now $A_\gamma = A$, and since $\varphi^A_e(x) \downarrow = 0$, there are plenty of ordinals $\sigma > \gamma > \beta$ for which $\varphi^{A_\sigma}_{e,\sigma}(x) \downarrow = 0$. But our algorithm runs on input 0 with no oracle, so its own halting time is clockable and greater than its output. Therefore there must exist a nonclockable stage σ between β and γ such that $\varphi^{A_\sigma}_{e,\sigma}(x) \downarrow = 0$. Let τ be the least clockable ordinal $> \sigma$. Then τ is a limit ordinal and $A_\sigma = A_\tau$, by our condition on changes to the enumeration of A, so this τ satisfies the lemma. $\Box$

Theorem 5 *There exist ORM-incomparable enumerable sets A and B of ordinals, both $\leq_{ORM} \emptyset^{\diamondsuit}$. It follows that $\emptyset <_{ORM} A <_{ORM} \emptyset^{\diamondsuit} (<_{ORM} \emptyset^{\blacklozenge})$, and likewise for B.*

PROOF. We build the ORM-enumerable sets A and B as follows, to satisfy the usual Friedberg-Muchnik requirements for all $e \in \omega$:

$$\mathcal{R}_e : \ (\exists x_e)[\varphi_e^A(x_e)\downarrow = 0 \text{ iff } x_e \in B] \qquad \mathcal{S}_e : \ (\exists y_e)[\varphi_e^B(y_e)\downarrow = 0 \text{ iff } y_e \in A].$$

These have the standard priority ranking: each $\mathcal{R}_e$ has higher priority than $\mathcal{S}_e$, which has higher priority than $\mathcal{R}_{e+1}$. At each stage σ we will have an approximation $x_{e,\sigma}$ to x_e, which converges to x_e as σ grows, and similarly for y_e.

Set $A_0 = B_0 = \emptyset$, and start with approximation $x_{e,0} = y_{e,0} = e$ to the witness elements x_e and y_e. We will redefine $x_{e,\sigma+1} \neq x_{e,\sigma}$ at only finitely many stages σ, namely those stages at which $\mathcal{S}$-requirements of higher priority than $\mathcal{R}_e$ act, and the same holds symmetrically for $y_{e,\sigma}$.

If τ is a limit ordinal, define $A_\tau = \cup_{\sigma<\tau} A_\sigma$, with $x_{e,\tau} = \lim_{\sigma\to\tau} x_{e,\sigma}$, and similarly for B_τ and $y_{e,\tau}$. (Notice that each $x_{e,\sigma}$ is eventually constant as σ approaches τ, so these limits exist.)

For successor ordinals $\tau = \sigma+1$, if σ is either unclockable or a successor ordinal itself, then we preserve the settings at stage $\sigma+1$: $A_{\sigma+1} = A_\sigma$, $x_{e,\sigma+1} = x_{e,\sigma}$, and so on. Only if σ is a clockable limit ordinal do we act at the successor stage $\sigma+1$, as follows.

At such a stage $\sigma+1$, we fix the highest-priority requirement, say $\mathcal{R}_e$, which requires attention, by which we mean that $x_{e,\sigma} < \sigma$ and $\varphi_{e,\sigma}^{A_\sigma}(x_{e,\sigma})\downarrow = 0$ and $x_{e,\sigma} \notin B_\sigma$. (The last condition means that we have not already used this witness element to satisfy requirement $\mathcal{R}_e$.) We act by enumerating $x_{e,\sigma}$ into $B_{\sigma+1}$, with $A_{\sigma+1} = A_\sigma$. We then set $x_{i,\sigma+1} = x_{i,\sigma}$ for all $i \in \omega$, and $y_{i,\sigma+1} = y_{i,\sigma}$ for all $i < e$. The lower-priority $\mathcal{S}$-requirements are injured at this stage, for we redefine

$$y_{e+j,\sigma+1} = y_{e+j,\sigma} + (x_{e,\sigma} + \sigma) + j + 1$$

for each $j \in \omega$. Notice that since $x_{e,\sigma} < \sigma$, we will have $B_{\sigma+1} \subseteq \sigma$ (by induction on the preceding stages). Induction also makes clear that the new witness elements $y_{e+j,\sigma+1}$ are all clockable, since the clockable ordinals are closed under addition and σ is clockable. If there is no requirement $\mathcal{R}_e$ which requires attention at stage $\sigma+1$, then we preserve all settings from stage σ.

If the highest-priority requirement requiring attention at stage $\sigma+1$ is an $\mathcal{S}$-requirement, say $\mathcal{S}_e$, we simply interchange A with B and the x-witnesses with

the y-witnesses in the above paragraph. At this stage, we preserve $y_{i,\sigma+1} = y_{i,\sigma}$ for all i, and $x_{i,\sigma+1} = x_{i,\sigma}$ for all $i \leq e$, with

$$x_{e+j,\sigma+1} = x_{e+j,\sigma} + (y_{e,\sigma} + \sigma) + j$$

for each $j > 0$ in ω, but not for $j = 0$. This reflects the fact that $\mathcal{R}_e$ has higher priority than $\mathcal{S}_e$. Otherwise, the entire process is symmetric in A and B.

The point of the redefinition of the y-elements when we satisfy $\mathcal{R}_e$ is that since the computation $\varphi_e^{A_\sigma}(x_{e,\sigma}) = 0$ converged in $\leq \sigma$ steps, the only oracle questions it can have asked involved membership of ordinals $\leq x_{e,\sigma} + \sigma$ in the oracle set. The elements which may later enter A are the witness elements $y_{i,\rho}$ at stages $\rho > \sigma$. By redefining $y_{e+j,\sigma+1} > x_{e,\sigma} + \sigma$ for all $j \geq 0$ at stage $\sigma + 1$, we ensure that any of these elements that later enters A will not change the oracle computation $\varphi_e^A(x_{e,\sigma}) = 0$, since the computation cannot have asked whether such large elements were in A. (It is possible for $y_{e+j,\rho}$ to be redefined yet again at a stage $\rho+1 > \sigma+1$, but our formula also ensures that it is always redefined to be larger than it had been before.) Of course, if a higher-priority y-witness element later enters A, it could change this computation, but the usual finite-injury argument shows that eventually we will reach a stage after which no higher-priority requirement acts again. So, using induction on the requirements according to their priority, we see that for every e, the witness elements $x_e = \lim_\sigma x_{e,\sigma}$ and $y_e = \lim_\sigma y_{e,\sigma}$ exist, and that if $x_e \in B$, then $\varphi_e^A(x_e)\downarrow = 0$, and symmetrically.

A further argument is necessary for the converse, using the Reflection Lemma 4. The point of restricting our construction to clockable stages was to ensure that every witness element $x_{e,\sigma}$ and $y_{e,\sigma}$ at every stage is a writable ordinal, i.e. equal to $\varphi_p(0)$ for some program p. The clockable limit stages were chosen precisely because, being clockable, they were writable, as are their successors and the stage 0. (Also, clockability is easier to check than writability!) Then, if we redefined any $x_{i,\sigma+1}$ at stage $\sigma + 1$, we set it equal to a sum of writable ordinals, and similarly for $y_{e,\sigma+1}$. So, by induction on stages, all witness elements are writable, and thus all elements of A and B are writable. But we can write the stage at which a writable ordinal enters a semidecidable set, so $A_\gamma = A$ and $B_\gamma = B$, as required by the Reflection Lemma.

Now suppose that x_e never entered B. Then for all sufficiently large clockable limit ordinals δ, $\varphi_{e,\delta}^{A_\delta}(x_e)$ either diverges or converges to a nonzero value, so by the Reflection Lemma, the full computation $\varphi_e^A(x_e)$ cannot converge to 0. Thus again x_e witnesses that φ_e^A does not compute B, so $B \not\leq_{ORM} A$. A symmetric result holds for y_e, so A and B are ORM-incomparable sets.

We have called this process a construction, and indeed it enumerated elements into A and B, without ever removing them. Nevertheless, it remains to show that A and B are actually ORM-enumerable, of course, since the description

above did not use ORM's. The heart of the proof of Theorem 5 is the following lemma.

Lemma 6 *There exists an ORM which decides, for arbitrary ordinal inputs α and σ, whether $\alpha \in A_{\sigma+1} - A_\sigma$; and similarly for B. We refer to the algorithms for these machines as the* entry algorithms *for A and B.*

PROOF. We use an ordinal stack machine, which is a special type of ordinal register machine. In an ordinal stack machine, along with finitely many registers, we have finitely many *stacks*, each of which (at any single stage of operation) consists of a descending (hence finite) sequence of ordinals. We can push a new ordinal from a register onto one of these stacks, provided that it is strictly less than all ordinals currently on the stack; and we can pop the smallest ordinal off of any stack and transfer it to a register, where it can be compared to the contents of other registers, etc., by the usual register operations. In [8], Koepke and Siders use the Cantor normal form of an ordinal to prove that the functions on ordinals computable by ordinal stack machines are precisely those computable by regular ORM's, so we may prove our lemma by giving two stack-machine programs (one for A, one for B) which answer the question for arbitrary α and σ.

Our stack machine accepts the inputs α and σ. It immediately pushes σ on top of its stage stack, and pushes the ordinal $(\omega^\sigma + \alpha)$ on top of its input stack. These will stay on these stacks for the rest of the operation of this machine, but occasionally the entry algorithm will call itself and push smaller ordinals above them, which are then popped off the stack when the subroutine ends. Of course, we have access to the value σ from the top of the stage stack any time we need it, and from the two stacks together we can also compute the value of α whenever we need it.

The reason for not simply pushing α onto the input stack is that the entry algorithm may later call itself, with inputs τ and β. We will ensure that $\tau < \sigma$, but we may have to allow $\beta \geq \alpha$, in which case we could not push β onto a stack above α. However, pushing $\omega^\tau + \beta$ onto the stack above $\omega^\sigma + \alpha$ will be legal, because we will always have $\beta < \sigma$ and $\tau < \sigma$.

We give the details for the entry algorithm which decides whether α entered B at stage $\sigma + 1$. The entry algorithm for A is quite similar, of course, and in fact is used by the algorithm for B. Given α and σ, we execute the following steps.

(1) If $\alpha \geq \sigma$, or if σ is not a clockable limit stage, then output 0. Otherwise, go on. (In our construction, a witness element α only enters B at successors of clockable limit stages $> \alpha$. These properties are indeed ORM-decidable.)

(2) For each $e < \omega$, check whether $x_{e,\sigma} = \alpha$. If such an e exists, then go on to Step 3. Notice that if e exists, then it is unique, and we can always use it in subsequent steps by using this same process to search for it again. If no such e exists, output 0. (Only witness elements for $\mathcal{R}$-requirements ever enter B. We prove in Lemma 7 that we can compute $x_{e,\sigma}$ uniformly from e and σ.)

(3) With the e from the previous step, we now simulate the operation of the program φ_e on input α with oracle A_σ for σ steps. Of course, we have no A_σ-oracle, so whenever our simulation asks whether some $\beta < \sigma$ lies in A_σ, we simply use the entry algorithm to check (for all τ with $0 \leq \tau < \sigma$, starting with 0) whether β entered A at stage τ. Notice that this is allowed by our stack machine, since each such τ and β is strictly less than σ, even though the β might be $> \alpha$. (The construction ensured that $A_\sigma \subseteq \sigma$, so if the simulation asks whether some $\beta \geq \sigma$ lies in A_σ, we answer "no" immediately, without using the entry algorithm to check.) Thus we can determine whether $\varphi_{e,\sigma}^{A_\sigma}(\alpha)\downarrow = 0$. If not, output 0; if so, go on.

(4) Now run the entry algorithm with α and with each stage $\tau < \sigma$, to see if $\alpha \in B_\sigma$. If so, output 0, since $\alpha \notin B_{\sigma+1} - B_\sigma$. If not, go on.

(5) Now compute $x_{i,\sigma}$ for each $i < e$, and run the entry algorithm with each such $x_{i,\sigma}$ and with stage σ to determine whether any higher-priority requirement than $\mathcal{R}_e$ enumerated any element into B at stage $\sigma + 1$. For this we do not push σ onto the stage stack, because to do so would be illegal. Nor do we pop it off that stack at the end, because we want it to stay there until the end of the run of the entry algorithm on α and σ. However, we do push $(\omega^\sigma + x_{i,\sigma})$ onto the input stack, which is legal since $x_{i,\sigma} < x_{e,\sigma}$ for $i < e$, and take it off again when we move on to the next i. If we find any such $x_{i,\sigma}$ which entered B at stage $\sigma + 1$, then output 0.

Also, we compute $y_{i,\sigma}$ for each $i < e$ and use the entry algorithm for A to determine whether any such $y_{i,\sigma}$ entered A at stage $\sigma+1$. Here we must be careful, because this run of the entry algorithm for B might have been called by the entry algorithm for A. So we check the top (i.e. smallest) element of the stage stack from the entry algorithm for A. If that element equals σ, then we leave it there, and just push $\omega^\sigma + y_{i,\sigma}$ onto the input stack, which is safe, because if the entry algorithm for A was already running and needed to know about $x_{e,\sigma}$ entering B, it must have been asking about an element $y_{j,\sigma}$ with $j \geq e > i$, so that $\omega^\sigma + y_{j,\sigma} > \omega^\sigma + y_{i,\sigma}$.

When the entry algorithm for A concludes, we find σ on top of its stage stack and consider the top two elements, say $\beta < \delta$, on its input stack. (If there is only one element β on the input stack, then we pop β off its stack and σ off its stack, leaving nothing on either stack.) Now β must equal $\omega^\sigma + y_{i,\sigma}$. Then we pop δ off the input stack and find the largest term of its Cantor normal form, say $\delta = \omega^\tau + \theta$ for some τ and θ. If $\tau = \sigma$, then we leave δ on top of the input stack and σ on top of the stage stack. If $\tau \neq \sigma$, then we leave δ on top of the input stack, but remove σ from the

stage stack. Thus, even though we left no specific indicator, when calling the entry algorithm for A, of whether we had added a new σ on top of the stage stack or not, we have still determined at the conclusion of that algorithm whether or not a new σ had been added, and if it had, then we have now removed it again.

Having completed the entry algorithm for A, we now know whether any $y_{i,\sigma}$ with $i < e$ entered A at stage $\sigma + 1$. If so, then output 0; if not, then output 1. This completes the entry algorithm for B.

(If any higher-priority requirement $\mathcal{R}_i$ or $\mathcal{S}_i$ enumerated an element into B at stage $\sigma + 1$, then $\mathcal{R}_e$ would not enumerate an element of its own. If not, then all conditions are satisfied for $\alpha = x_{e,\sigma}$ to enter B at stage $\sigma + 1$.)

We define an analogous entry algorithm to check whether an arbitrary α entered A at an arbitrary stage $\sigma + 1$, of course. Indeed, these two algorithms call each other in certain cases, as detailed in Item (5). Because $\mathcal{R}_e$ has higher priority than $\mathcal{S}_e$, the entry algorithm for A must check in Item 5 whether any $x_{i,\sigma}$ with $i \leq e$ entered B at stage $\sigma + 1$, rather than just checking for $i < e$. Otherwise, the entry algorithms are symmetric in A and B.

Both entry algorithms are computable by stack machines, and the proof that they give the correct answer is a fairly simple double induction, first on stages σ, and for each individual stage, on inputs $\alpha < \sigma$. The one twist to be noted is that we promised a separate algorithm to compute $x_{e,\sigma}$ for arbitrary e and σ; and since this algorithm is used in the entry algorithm, it can only call the entry algorithm for smaller stages.

Lemma 7 *There are ORM's which compute $x_{e,\sigma}$ and $y_{e,\sigma}$, uniformly in e and σ, We refer to their programs as the* witness algorithms *for A and B. These ORM's use the entry algorithm from Lemma 6, but they only push stages $< \sigma$ onto the stage stack. (That is, they only ask whether elements entered A or B at stages $\tau + 1$ with $\tau < \sigma$.)*

PROOF. The witness algorithm is allowed to call itself, but only with descending stages, just like the entry algorithms. To compute $y_{e,\sigma}$, we start with $e = y_{e,0}$ in the output register of a stack machine. Then go through all ordinal stages $\tau = 0, 1, \ldots$. As long as $\tau + 1 \leq \sigma$, we use the witness algorithm to compute $x_{0,\tau}, \ldots, x_{e,\tau}$ and then the entry algorithm to check whether any $\beta = x_{i,\tau}$ with $i \leq e$ entered B at stage $\tau + 1$. If so, then $y_{e,\tau+1}$ will have been redefined to equal $y_{e,\tau} + \beta + \tau + (e - i) + 1$, so we write this new value in the output register in place of $y_{e,\tau}$. Otherwise we leave the output register as it is. When $\tau + 1$ finally equals $\sigma + 1$, we halt, and the value in the output register will be $y_{e,\sigma}$.

Notice that the corresponding routine for computing $x_{e,\sigma}$ asks only about elements $y_{i,\tau}$ with $i < e$. This mirrors the priority ranking of the requirements, and is important for seeing that the inductive argument for correctness of this algorithm is well-founded. This proves Lemma 7, and also Lemma 6. $\square$

With Lemma 6, it is clear that the sets A and B are ORM-enumerable. A, for instance, is the domain of the ORM-computable function which, on input α, starts with $\sigma = 0$, asks for each σ in turn whether α enters A at stage $\sigma + 1$, and halts, putting α in the domain, if it ever receives a positive answer.

ORM-enumerability immediately implies that $A \leq_{ORM} \emptyset^{\blacklozenge}$, but we claim that $A \leq_{ORM} \emptyset^{\diamond}$ as well. Given any ordinal x, we check first whether x is clockable. If not, then $x \notin A$. (When choosing witness elements, we made sure they were all clockable.) Otherwise, there is an ORM program q which writes x and then gives x as an input to the ORM-computable function whose domain is A. We can compute this q uniformly from x, and $x \in A$ iff $q \in \emptyset^{\diamond}$. The same proof works for B, so we have a pair of ORM-incomparable semidecidable sets below $\emptyset^{\diamond}$, as Theorem 5 claimed. $\square$

3 Ordinal Time Turing Machines

An *ordinal Turing machine*, or OTM, runs a finite program, using a single tape (equivalently, finitely many tapes) of ordinal length. That is, this tape has one cell for each ordinal, on which a single bit (0 or 1) can be written. The head moves left and right on the tape just as a finite-time Turing machine does, reading cells and writing over them, with an exception occurring when the head attempts to move left from a cell with a limit ordinal position. In this case, we write the current cell position in Cantor normal form and subtract 1 from the rightmost nonzero coefficient to get the new ordinal position of the head. (A simpler alternative is for the head to move all the way back to the zero-ordinal cell if it is instructed to move left when sitting on a limit-ordinal cell. This gives equivalent computing power.) At limit-ordinal stages, every cell sets itself to the liminf of the bits it has held up to that stage, and the machine enters a special limit state (or equivalently, enters the liminf of the states it has been in up till that stage). The input to an OTM can be any set of ordinals, described by an appropriate string of bits on the tape. However, if we wish to consider only ordinal inputs, we usually represent input α by setting the first α bits on the tape to 1 and the rest to 0. Outputs are evaluated similarly, depending on whether we see them as ordinals or sets of ordinals. For more details about OTM's, see [6].

The original conception of machine computation in infinite time was the *infi-*

nite time Turing machine of Hamkins and Lewis in [2], which runs in ordinal time but still has a tape of length ω. Such a machine can be simulated by an OTM in which the limit-state instruction always sends the head back to the zero-ordinal cell.

Theorem 8 *Let f be any partial function from ordinals to ordinals. Then f is computable by an ordinal Turing machine iff f is computable by an ordinal register machine. Moreover, the program for the ORM is computable in finite time from the program for the OTM and vice versa.*

PROOF. The proof of the converse, where f is assumed to be computed by an ORM, is straightforward, especially if one allows the OTM to have as many tapes as the ORM has registers. (This is equivalent to a single-tape OTM, of course.) The ORM operations can all be simulated by the OTM, and so the ORM program translates easily (by a finite-time computation) into an OTM program.

For the more substantial direction, assume that we have a program for an ordinal Turing machine with a single one-way ordinal tape, such that, if the tape begins with the first ξ-many cells set to 1 and the rest to 0 (representing the input ξ), then the OTM halts iff $\xi \in \mathrm{dom}(f)$, with the first $f(\xi)$ cells set to 1 and the rest to 0 when (and if) it enters the halting state. To see that some ORM also computes f on ordinals, we need the following lemma.

Lemma 9 *There are two ordinal stack machines $\mathcal{M}$ and $\mathcal{N}$ such that on input $\langle \delta, \sigma, \xi \rangle$, $\mathcal{M}$ outputs the value written in cell δ on the OTM tape at stage σ when the OTM runs with input ξ, and $\mathcal{N}$ outputs the pair $\langle q, \beta \rangle$, where β is the position of its reading head of the OTM and q is the state it is in after σ steps on input ξ. The programs of $\mathcal{M}$ and $\mathcal{N}$ are computable in finite time from the OTM program.*

PROOF. $\mathcal{M}$ and $\mathcal{N}$ will be allowed to call each other, but we will make sure that they only push smaller values onto each other's stacks when doing so, of course. Each machine has a single *input register* containing the value ξ, since this will stay fixed throughout their computations. Each machine also has a *stage stack*, and a *cell stack*, and $\mathcal{M}$ has two other stacks s_1 and s_2, while $\mathcal{N}$ has s_1, s_2, and s_3.

To begin the run of the machine $\mathcal{M}$, we consider δ. If $\delta \geq \max(\xi, \sigma)$, then we output 0 immediately, since the reading head of a Turing machine, starting on the leftmost cell, cannot reach cell δ in $< \delta$ steps. Likewise, if $\sigma \leq \delta < \xi$, we output 1, because then cell δ began with a 1 in it and cannot have been reached by the head within σ stages. Otherwise we have $\delta < \sigma$. In this case

we push σ onto stage stack of $\mathcal{M}$, and $\omega^\sigma + \delta$ onto its cell stack, allowing us to recover each of σ, δ, and ξ (from the input register) whenever we need them.

We check whether σ is a successor ordinal, determining its immediate predecessor τ if it is one. If so, then we run $\mathcal{N}$ on the value $\langle \delta, \tau, \xi \rangle$ to determine the state q of the OTM after stage τ and the position β of its head at that stage, If $\beta \neq \delta$, we simply run $\mathcal{M}$ on input $\langle \delta, \tau, \xi \rangle$ and output this value, since the content of cell δ cannot change between stages τ and $\tau + 1 = \sigma$ unless the head is on cell δ at stage τ. If $\delta = \beta$, we push $\omega^\sigma + q$ onto stack s_1, run $\mathcal{M}$ on input $\langle \delta, \tau, \xi \rangle$ to determine the content of cell δ at stage τ, get the state q back from stack s_1, consult the program for the OTM to determine what value i the OTM wrote on cell δ at step σ (i.e. going from stage τ to stage σ), and output i.

If σ was a limit ordinal, then we push ω^σ onto the stack s_1 and run the following loop:

> Determine the τ such that s_1 contains $\omega^\sigma + \tau$. Halt if $\tau = \sigma$, and otherwise run $\mathcal{M}$ on input $\langle \delta, \tau, \xi \rangle$, yielding an output i. Pop the current value off stack s_2, pushing $\omega^\sigma + i$ onto s_2 in its place, and pop the current value $\omega^\sigma + \tau$ off stack s_1, pushing $\omega^\sigma + \tau + 1$ onto s_1 in its place. Then repeat the loop.

When this loop stops, pop the top value $\omega^\sigma + i$ off stack s_2, recover i from it, and output i. At each stage $\tau \leq \sigma$, the smallest element on stack s_2 was $\omega^\sigma + i$, where i was the bit written in cell δ at stage τ in the operation of the OTM. (By induction, this holds at limit stages as well as successor stages, since OTM's and ORM's both take liminfs at limit stages.) So the i we recovered above was the correct output for $\mathcal{M}$.

The machine $\mathcal{N}$ actually never uses the δ from its input $\langle \delta, \sigma, \xi \rangle$. It pushes σ onto its stage stack and checks whether σ is a successor ordinal. If $\sigma = \tau + 1$, say, then it runs machine $\mathcal{N}$ on input $\langle \delta, \tau, \xi \rangle$ to determine the state q and head location β of the OTM at stage τ. Pushing $\omega^\sigma + q$ onto its stack s_1 and $\omega^\sigma + \beta$ onto s_2, it then runs $\mathcal{M}$ on input $\langle \beta, \tau, \xi \rangle$ to find the content i of cell β at stage τ, recovers q from s_1, and consults the OTM program to determine operation of the machine at stage σ: since it was in state q and reading an i at stage τ, the program tells what state q' it must enter at stage σ, and also tells whether the head will move left or right. If the head moves right, we output $\langle q', \beta + 1 \rangle$. If the head moves left, then we determine the Cantor normal form of β, chop off the last term, leaving some $\gamma < \beta$ (or 0 if $\beta = 0$), and output $\langle q', \gamma \rangle$. (For the simpler form of OTM, of course, we just output $\langle q', 0 \rangle$, since those OTM's automatically return their head to cell 0 in this situation.)

If σ is a limit ordinal, then the operation of $\mathcal{N}$ is very similar to that of $\mathcal{M}$ in the corresponding situation. Instead of pushing $\omega^\sigma + i$ onto stack s_2, however,

we push $\omega^\sigma + q$ onto s_2 and $\omega^\sigma + \beta$ onto s_3, where $\langle q, \beta \rangle$ is the output of $\mathcal{N}$ on input $\langle \delta, \tau, \xi \rangle$. When the loop ends, we pop the new values $\omega^\sigma + q$ off s_2 and $\omega^\sigma + \beta$ off s_3, recover q and β from them, and output $\langle q, \beta \rangle$.

A careful reading shows that each machine $\mathcal{M}$ and $\mathcal{N}$, on input $\langle \delta, \sigma, \xi \rangle$, only calls itself or the other machine with inputs where the middle term τ is $< \sigma$. This means that all values pushed onto stacks in the preceding instructions really are smaller than the values currently on top of those stacks. It also shows that the obvious inductive argument for the correctness of the outputs of $\mathcal{M}$ and $\mathcal{N}$ is well-founded. This completes the proof of Lemma 9. $\square$

The ORM which computes f is easily built from $\mathcal{M}$ and $\mathcal{N}$. Starting with an input ξ in its input register, it sets its *stage register* r_0 to 0, then runs the following loop:

> Run $\mathcal{N}$ with input $\langle 0, \sigma, \xi \rangle$, where σ is the value in r_0 and ξ is the value in the input register. If the output of $\mathcal{N}$ indicates that the OTM is not in the halting state at stage σ, then increment r_0 and repeat. If the OTM is in the halting state, set the output register r_1 to 0 and run the following loop. Run $\mathcal{M}$ on input $\langle \delta, \sigma, \xi \rangle$, with ξ still from the input register, δ from r_1, and σ from r_0. If the output of $\mathcal{M}$ is 1, then increment r_1 and repeat; otherwise halt.

So if the OTM never halts on input ξ, neither does our ORM. If the OTM halts at some stage σ, then according to our conventions, its tape contains $f(\xi)$-many consecutive 1's at that stage, followed by 0's. In this case, the second loop of the ORM keeps incrementing the output register r_1 until r_1 contains $f(\xi)$, at which point $\mathcal{M}$ indicates that the next cell on the OTM tape at stage σ is a 0, so the ORM halts with $f(\xi)$ written in r_1.

Finally, it is clear that the program for our ORM uses the OTM program practically verbatim in two spots (namely the successor-ordinal cases for $\mathcal{M}$ and $\mathcal{N}$, which are where the OTM instructions matter) and does not depend on the OTM program anywhere else. So our ORM program is uniformly finite-time computable from the OTM program, as claimed. $\square$

The obvious next question has to do with runtimes. If the OTM in question is an α-Turing machine, i.e. halts in fewer than α steps (or not at all) on inputs less than α, can we compute the same function f using an α-register machine? (And conversely?) An α-register machine halts in fewer than α steps (or not at all) and never writes any ordinal $\geq \alpha$ in any register. The α-Turing machines, of course, cannot reach any cell with a number greater than or equal to α in fewer than α steps, so it is not necessary to state any restriction on memory for α-TM's; and it is unnecessary for α-RM's as well in the case where α is

a power of ω. (Such powers $\alpha = \omega^\gamma$ are characterized by the property that $(\forall \beta < \alpha)\beta + \alpha = \alpha$, so on input $\beta < \alpha$, the input register cannot reach α in fewer than α steps.) Koepke has shown in [7] that for admissible α, the α-Turing machines and the α-register machines both compute precisely the α-partial recursive functions, as defined in α-recursion theory, for instance in [10]. So for these α, the answer is yes.

4 Softer proofs of the results

In the preceding arguments, we provided detailed ORM procedures for constructing a solution to Post's Problem and simulating an ordinal Turing machine. We are pleased to have done so, and we believe that the procedure helps illustrate several useful techniques of ORM computation. Nevertheless, this part of the argument can be completely eliminated by making use of the main theorem of [8]. The structure of our argument for Post's Problem, for example, was first to provide a set-theoretical definition of the sets A and B in terms of their approximations A_σ and B_σ, which ensured that A and B would be ORM-incomparable, and then to provide a detailed ORM algorithm to decide the entry problems $\alpha \in A_{\sigma+1} - A_\sigma$ and $\alpha \in B_{\sigma+1} - B_\sigma$, which ensured that A and B would be ORM-enumerable. The point we would like to make now is that once we have given the initial set-theoretic construction of the sets A_σ and B_σ, we can simply observe that this construction can be carried out inside Gödel's constructible universe L, in such a way that the construction is absolute to initial segments of L. That is, if some level of the constructibility hierarchy L_η satisfies that an ordinal α has entered A at stage σ, then this is truly the case. Therefore, in order to determine whether or not α enters A at stage σ, we need only search for an ordinal η such that L_η satisfies the set-theoretical assertion "α enters A at stage σ". But the question of whether L_η satisfies a given set-theoretical assertion $\varphi(\alpha, \sigma)$ is uniformly ORM-decidable from input $(\eta, \alpha, \sigma, \ulcorner\varphi\urcorner)$, by the main result of [8]. Consequently, both A and B are ORM-enumerable, as desired. A similar analysis applies to Theorem 8.

In addition, we would like to mention that an even softer argument for Post's Problem can be made by appealing to the Sacks-Simpson [11] solution of Post's problem in α-recursion theory. It has been observed by Koepke in [7], after the Bonn International Workshop on Ordinal Computability 2007, that for any admissible ordinal α, the subsets of α that are computable in α-recursion theory are exactly the sets that are ORM-computable with parameters in time less than α. Using the admissible ordinal γ, the supremum of the ORM-clockable ordinals, one may now transfer the solution of Post's problem from γ-recursion theory over to ordinal register machines. Koepke provides the details of this idea in [7] in the case of ordinal Turing machines, but explains how this argument also applies to ordinal register machines.

References

[1] R.M. Friedberg, Two recursively enumerable sets of incomparable degrees of unsolvability, *Proc. Nat. Acad. Sci. (USA)* **43** (1957) 236–238.

[2] J.D. Hamkins & A. Lewis, Infinite time Turing machines, *Journal of Symbolic Logic* **65** 2 (2000) 567–604.

[3] J.D. Hamkins & A. Lewis, Post's problem for supertasks has both positive and negative solutions, *Arch. Math. Logic* **41** (2002) 507–523.

[4] J.D. Hamkins & R.G. Miller, Post's Problem for ordinal register machines, in: eds. B. Cooper, B. Löwe, & A. Sorbi, *Computation and Logic in the Real World – Third Conference on Computability in Europe, CiE 2007*, *Lecture Notes in Computer Science* **4497** (Springer-Verlag, Berlin, 2007) 358–367.

[5] L. Harrington & R.I. Soare, Post's Program and incomplete recursively enumerable sets, *Proc. Nat. Acad. Sci. (USA)* **88** (1991) 10242–10246.

[6] P. Koepke, Turing computations on ordinals, *Bulletin of Symbolic Logic* **11** 3 (2005) 377–397.

[7] B. Dawson, P. Koepke, & B. Seyfferth, Ordinal machines and admissible recursion theory, submitted for publication.

[8] P. Koepke & R. Siders, Register computations on ordinals, submitted for publication.

[9] A.A. Muchnik, On the unsolvability of the problem of reducibility in the theory of algorithms, *Dokl. Akad. Nauk SSSR, N.S.* **109** (1956) 194–197 (Russian).

[10] G.E. Sacks, *Higher Recursion Theory* (Berlin, Springer-Verlag, 1990).

[11] G.E. Sacks and Stephen G. Simpson, The α-finite injury method, *Annals of Mathematical Logic* 4 (1972) 343–367.

[12] R.I. Soare, *Recursively Enumerable Sets and Degrees* (New York, Springer-Verlag, 1987).

[13] P. Welch, The lengths of infinite time Turing machine computations, *Bulletin of the London Mathematical Society* **32** 2 (2000) 129–136.