Minimality Considerations for Ordinal Computers Modeling Constructibility

Peter Koepke, Ryan Siders *

Mathematisches Institut, Universität Bonn, Germany

Department of Mathematics and Statistics, University of Helsinki, Finland

Abstract

We describe a simple model of ordinal computation which can compute truth in the constructible universe. We try to use well-structured programs and direct limits of states at limit times whenever possible. This may make it easier to define a model of ordinal computation within other systems of hypercomputation, especially systems inspired by physical models.

We write a program to compute truth in the constructible universe on an ordinal register machine. We prove that the number of registers in a well-structured universal ordinal register machine is always ≥ 4, greater than the minimum number of registers in a well-structured universal finite-time natural number-storing register machine, but that it can always be kept finite. We conjecture that this number is four. We compare the efficiency of our program which computes the constructible sets to a similar program for an ordinal Turing machine.

Key words: hypercomputation, well-structured programming, set theory, constructibility, ordinal arithmetic, abstract computability

1 Computation at a limit time – continuity and loops are enough.

Let an ordinal computer be a register machine in the sense of [8], storing ordinals and running for ordinal time. Abstract computation, which puts non-integer register values into the registers of a computer, was pursued in [2] and recently in [9]. Ordinal runtimes were described in [1] and [4] and [5], among others. In [6] we related that model of computation to set theoretic notions including the recursive truth predicate and the theory of sets of ordinals. This

* Corresponding author, email: ryan.bissell-siders@helsinki.fi

paper presents ordinal computation from a machine-focused point of view, and considers the structure of algorithms, direct limits, and complexity.

The active command line cannot be a continuous function of time at a limit time, and some registers will not be continuous at limit times. Legal programs, whose `if`-switches are monotonic or are recently computed from monotonically increasing variables, compute their results without any assumption – beyond continuity – on how register values behave at limit times. Well-structured programs (written as loops of loops as in [3]) compute their results without any assumption on how the command control behaves at a limit time, other than that control does not pass out of a loop until the loop's condition is met. We define ordinal register machines to include illegal and illstructured program, and in Claim 7 we present their "wellstructured," "legal" version definition and prove, by the end of this section, that these two definitions are equivalent.

Definition 1 *An ordinal register machine contains a finite number of registers, each of which can store an ordinal. A program is a numbered list of commands, each of which has one of the following three forms:*

- `Zero(`x`)` *: Erases the content of register* x*, leaving* 0*.*
- x `++` *: Increments the value of register* x*.*
- `if` $x = y$ `goto` i `else` j *: switches line control.*

The value of a register x *must be a continuous function of time, over any interval of time in which the command* `Zero(`x`)` *is not executed. In addition, the state (register values and active command) obeys the following rules at limit times:*

(1) If the command `Zero(`x`)` *is executed at each time* $t \in T$*, then the value of register* x *at time* $\sup T$ *is zero.*
(2) At a limit time λ*, command passes to the minimum of the commands which have been active cofinally often before* λ*.*

The last rule is the same as in [5]. In [4], the active command is stored in binary memory; each bit becomes the lim-sup of its previous values at a limit time. On such a machine, we can code the active command so that it becomes the lim-sup or the lim-inf of the commands active previously. However, for well-structured programs, we can replace all requirements, beyond continuity, on how registers behave at limit times by the requirement that any repeating loop should begin again, at a limit time, by first checking the conditional, and then executing the loop again, and so on.

Definition 2 *A program is well-structured if* `...` `goto` `...` *switches are only used to model the following two commands:*

- `if` $x = y$ `(loop)`.
- `for` x `to` y `(loop)` *where the command* `Zero(`x`)` *is not among the instructions in the loop, or*
- `while` $x \leq y$ `(loop)`, *where the command* `Zero(`x`)` *is not in the loop, and where it is provable that* x *will be incremented at least once during the loop.*

The loop `for` x `to` z `(`L`)` *is defined in terms of* `goto` *as* `1. if` $x > z$ `goto 2;` L`;` x`++; if` $x \leq z$ `goto 1; 2.`

The command `for` x `from` 0 `to` z `(`L`)` *is* `Zero(`x`); for` x `to` z `(`L`).`

The command `while` $x \leq y$ `(`L`)` *is defined from* `goto` *as* `1. if` $x > z$ `goto 2;` L`; if` $x \leq z$ `goto 1; 2...` *That* x`++` *will be executed at least once during the loop* L*, and that* `Zero(`x`)` *is not allowed to appear within* L *assures that* x *will grow monotonically, and therefore it will eventually reach or exceed the loop bound.*

The loop `while` $x = 0$ `(`L`)`*, in which* x *does not necessarily increment during* L*, is not considered wellstructured programming. Since* x *need not increment, the loop could be repeated forever, unlike loops which halt at the fixed points of the normal functions computed by wellstructured programs.* `while(`$x = 0$`)` `(`L`)` *is defined from* `goto` *as* `1. if` $x > 0$ `goto 2;` L`; if` $x = 0$ `goto 1; 2...` *where* L *does not necessarily increment* x*.*

The well-structured programs form the smallest set of programs containing the basic commands `Zero(`x`)` *and* x `++` *of definition 1 for all register values of* x *and* y*, and closed under concatenation and repeating any sequence of wellstructured programs within a wellstructured loop.*

A `for` loop increments its index during each loop, and halts when $x = z$. To make this act like traditional `for` loops, we say that the loop is executed one more time once $x = z$ is reached. The for loop must be programmed on the ordinal register machine so that the conditional is checked first.

On the other hand, a program which increments x and then checks whether $x = y$, and halts if so, and then increments y, and repeats those three steps, will never halt, unless at some limit time, control passes to "`if` $x = y$" or "y`++`" rather than to "x`++`." For instance, if the condition on the loop is $f(x, y, z) < g(x, w)$, then at the end of the loop, we compute $u = f(x, y, z)$ and $v = G(x, w)$, and the minimum instruction in the loop is to compare whether $u < v$. When $u = v$, the loop ends, after executing one more time.

However, checking the conditions which could terminate a loop, immediately on reaching any limit times, leads to:

Lemma 3 *Well-structured programs halt.*

Proof: By induction on loops, considered as subprograms for which the lemma is proved. During the execution of the loop `for` x `to` y `(loop)`, the register values are all bounded by $c+$ time, for any c which bounds the initial values of the registers. The absolute number of timesteps used, limited by the length of the program and the register values, is a normal function of the loop index, register x, and so at fixed points of this function, the value of x is time. Therefore, on or before the first such fixed point, the condition $x \geq z$ is met. □

Any register which is not erased for a long time, like the index of a loop, becomes frequently equal to the value of absolute time. On the other hand,

Lemma 4 *If registers* $\{x_i : i < \omega\}$ *are erased cofinally often before limit time* λ*, then at some time* $\leq \lambda$*, all the registers are simultaneously zero.*

Proof: Let π_0 and π_1 be functions from ω to ω, such that (π_0, π_1) enumerates $\omega \times \omega$. Let $t_0 < \lambda$. If t_{n-1} has been defined, then let t_n be the next time after t_{n-1} when register $x_{\pi_0(n)}$ is erased. For each i, $\sup\{t_n : n < \omega\} = \sup\{t_n : n < \omega$ and $\pi_0(n) = i\}$, so at that time, register i contains the value zero. □

During the execution of a loop, some registers will be erased and others (at least the indices of the loop) will never be erased, from which we define

Definition 5 *Within a loop, call register* x scratch *if the command* `Zero(`x`)` *occurs; if not,* monotone *if the command* x`++` *occurs; and* constant *otherwise.*

In programs presented in this paper, we will use the symbols `MON` or `SCR` to define a variable to be monotone or scratch in this sense.

We want to make the following program illegal:

- `for` i `from` 0 `to` ω `(Zero(`x`); `x`++);`
- `if` $x = 0$ `(Zero(`y`))`

because it tests the limit of a noncontinuous register value. We want to call x a scratch variable and prevent a variable which, like x, has been erased infinitely often from appearing in the conditional of an `if`-switch until after it has been erased again. Suppose f and g are normal functions of two variables, that can be computed without using the commands `Zero(`x`)` or `Zero(`y`)`. The following program should be legal:

- `for` i `from` 0 `to` ω `(`$x =$ `f(`x, y`); `$y =$ `g(`x, y`));`
- `Zero(`v`);`
- `for` i `from` 0 `to` y `(for` u `from` 0 `to` v `(`v`++));`
- `if` $v = f(x, x)$ `(Zero(`z`))`

We might abbreviate the last two lines as `if` $2^y = f(x,x)$ `(Zero(`z`);)`. In case v were larger than 2^y, we had to erase v before we could increase it monotonically to $v = 2^y$. This should be legal because y is monotone and v depends only on y.

Definition 6 *If a program contains the following:*

- *a loop A which contains the command* `Zero(`x`)`*; let X be the first command in the loop A (writing the program using* `increment`*,* `goto`*, and* `zero` *commands, X appears as the earliest command),*
- *a path B from the command X to the command Y,*
- *Y performs a switch on the variable x, and*
- *in the path B the command* `Zero(`x`)` *doesn't occur,*

then the the program is illegal. [1]

An illegal program can test the limit of a discontinuous variable. This is because the program could loop infinitely often through the loop A, each time possibly zeroing x, and then switch on the value of x. The switch would then notice whether x is zero after being zero'd (and perhaps increased) infinitely often. We want to write programs independent of the limiting behavior of scratch variables, i.e., independent of condition 1 in definition 1. Using this notion, we can say formally that a scratch variable is not legal in an `if` conditional immediately after it has been incremented and erased infinitely often, but that immediately after it has been erased one more time, it becomes *legal* for use in an `if` conditional. Now we have replaced both conditions 1 and 2 of definition 1 with restrictions on how programs are written, so it turns out that those conditions are not necessary to the proper working of an infinitary machine.

Claim 7 *The class of computable functions remains the same if, in definition 1, we require the program to have the form* `while(`$x = 0$`) loop`*, where* `loop` *is a wellstructured program obeying the following two programming techniques:*

- *Explicitly empty all scratch registers at the beginning of the loop.*
- *Registers used in a switch or the conditional of a loop should depend in a wellfounded way on monotone registers.*

The assumptions we need on how a state behaves at a limit time can be relaxed from conditions 1 and 2 in definition 1 to:

(1) at a limit time, a wellstructured program evaluates the (unique) active loop, and determines whether to continue looping, and

[1] We thank the anonymous referee for suggesting this definition, which simplifes and corrects an earlier definition of illegal programming.

(2) monotone register values pass continuously to their limits (i.e., don't jump) at limit times.

Wellstructured programs always halt, since they compute normal functions and halt when the loop bound is reached by the loop index. Nonwellfounded programs can certainly perform unbounded search. Hence, the `while` loop in the statement of this claim is necessary.

During the rest of this section, we will prove the claim as a generalization of the theorem in [3] reducing all branching programs to loops. That theorem applies only to finite-time computers storing ordinals. However, the theorem still applies if we make the signature (the set of functions and predicates that can be performed instantaneously in a particular model of computation, as in [9] page 322.) include ordinal arithmetic and Gödel pairing.

A finite-time ordinal-storing register machine with only the successor in its signature (an ORM operating for finite time has the successor and Zero in its signature) cannot do arithmetic on its memory elements. The functions of addition, of finding the predecessor of a successor, etc., all take infinite time. However, it is clear that an ordinal register machine can perform these operations of arithmetic, since ordinal addition is iterated succession, multiplication is iterated addition, and exponentiation is iterated multiplication (see details before Lemma 12). Further, the Gödel pairing function, sending (α, β) to the order type of $(\alpha \times \beta, <^g)$, where $<^g$ is the ordering that first compares maxima, then compares lexicographical order (more details in [6], section 2) is also clearly computable by an ordinal register machine that enumerates $\alpha \times \beta$ in the desired order and increments $G(\alpha, \beta)$ with each step.

Definition 8 *Let G be the pairing function taking (a, b) to the order type of pairs $(c, d) <^g (a, b)$, where $(c, d) <^g (a, b)$ iff $\max(c, d) < \max(a, b)$, or $\max(c, d) = \max(a, b)$ and $c < a$, or $\max(c, d) = \max(a, b)$ and $c = a$ and $d < b$.*

It is clear how to program the preceding definitions, so ordinal register machines surely compute the signature $\Sigma = (Ord, 0, 1, +, \times, \exp, G, G^{-1})$. Ordinal computers performing finite sequences of operations in that signature form the set of **While**-computable functions over Σ, defined in [9] page 323. That is equivalent to any other reasonable notion of what a finite-time computer could compute, with ordinals in storage and oracles for the functions in Σ.

Definition 9 *Call a finite-time register machine storing ordinals and able to compute the functions in signature Σ in one step an abstract ordinal computer.*

Since the signature Σ can code sequences of ordinals as a single ordinal, and since ordinals contain the natural numbers, many natural notions of abstract computability agree over the ordinals, including the interesting machine mod-

els described in section 8 of [9].

Proof (of claim 7): Suppose a model of ordinal computation is proposed, so that on input x it produces output y iff $\phi(x, y, \alpha_0...\alpha_n)$ holds, where ϕ is a Δ_1 concept of set theory. For instance, ϕ might say that there is a computation that starts with x (and the parameters), proceeds according to Δ_0 rules (where $\psi(x, y)$ is Δ_0 if all quantification is bounded to x and y) and ends with a designated output register holding the value y. Then ϕ has a Σ_1 representation. If *any* computation which starts with x and proceeds legally must end with y, then the model has a Δ_1 description. For instance, any reasonable variation on our definition of ordinal computer has a Δ_1 description. We mean to show that all of these can be modeled on our computer. To determine the truth of ϕ, we search through L to find either an example that proves ϕ in its Σ_1 form, or a counterexample that disproves ϕ in its Π_1 form. The `while` loop in the claim 7 can perform this search, if the following lemma holds.□

Lemma 10 *Wellstructured programs can determine the truth of any Δ_0 sentence, with constant symbols referring to ordinals, of ZFC.*

Proof: Fix an enumeration of formulas with ordinal parameters to prove this lemma by induction. The abstract ordinal computers in definition 9 are Church-Turing complete, so they can compute syntactic operations on formulas, in their codes as ordinals (we will mention this again in definition 16). For instance, we can make the description of ϕ be shallowly accessible in the ordinal coding of ϕ and its parameters $\alpha_1...\alpha_n$ as $G(n_\phi, G(\alpha_0, G(...)))$, where n_ϕ is the Gödel number of the formula. We can choose that the operations $\neg, \wedge, \vee$ increase the Gödel number of a formula, so that to prove the lemma by induction, we only have to deal with formulas $\exists z < x\psi$ or $\forall z < x\psi$. The program corresponding to ϕ has an outer loop `for` z `from` 0 `to` x `loop`, where the loop pushes z into the stack of variables, and then runs the program corresponding to ψ to determines whether ψ holds for that particular value of z. □

2 A Universal Ordinal Register Machine Program

In this section we write a universal program. This improves on Lemma 9 which found a wellstructured program to decide each bounded formula ϕ. The universal L-program reads a code for ϕ and its parameters as input, and determines the truth of ϕ in time at most ordinal-exponential in the size of those parameters. To be precise about the size of the parameters, the reader may wish to check that $G(n_\phi, G(\alpha_0, G(\alpha_1, ...)))$ is $\leq$ the first ordinal of the form ω^{ω^α} which is larger than all of the α_i.

Lemma 11 *$G(\gamma,\gamma)=\gamma$ iff γ is a $\times$-closed ordinal.*

Proof: Exercise. Hint (only if) Prove $G(\alpha\times\beta,\alpha\times\beta)>\alpha\times\beta$ unless α or β is 1, by finding a large ordinal product contained in the order type of $<^g$ in definition 8. Hint (if) Prove by induction that if α-many Cantor-Bendixon derivatives (which "derivative" eliminates all the successor elements) reduce γ to a finite set, then $\alpha\times 2+1$-many Cantor-Bendixon derivatives reduce $G(\gamma,\gamma)$ to the empty set. As a result, every element of $|G(\omega^{\omega^\alpha},\omega^{\omega^\alpha}),<^g|$ is eliminated by $<\omega^\alpha$-many derivatives. □

Proof (only if): We prove $G(\alpha\times\beta,\alpha\times\beta)\geq\alpha^2\times(-1+\beta)$ (by $-1+\beta$ we mean the ordinal which is β if β is infinite, and $\beta-1$ if β is finite... it is obtained during our proof as a set isomorphic to β, but missing its first element, hence we write it in this way). Label the elements of $\alpha\times\beta$ as $\{(a,b):a\in\alpha;b\in\beta\}$. The ordering on the ordinal $\alpha\times\beta$ is $<_l$, the reverse lexicographical ordering: $(a,b)<_l(a',b')$ if $b<b'$ or $b=b'$ and $a<a'$. Now for each $b\in\beta$, b not maximal in β, $G(\alpha\times\beta,\alpha\times\beta)$ gives $S_b:=\{((a,b),(a',b+1):a,a'\in\alpha\}$ its lexicographical order because the pair $((a,b),(a',b+1))$ achieves its maximum on its second element. G orders S_b as $\alpha\times\alpha$, there are at least $-1+\beta$ many sets S_b, and G orders the sets S_b in the same order as β orders the pairs $(b,b+1)$. Proof (if): The $\times$-closed ordinals are the ordinals ω^α for various α. We proceed by induction on α. If α is a successor, then $G(\omega^\alpha,\omega^\alpha)=\sum_{n<\omega}G(\omega^{\alpha-1}\times n,\omega^{\alpha-1}\times n)$. Taking Cantor-Bendixon derivatives (passing from a set to the set of its limit points) of that order type α many times leaves the emptyset, so the order type is $\leq\omega^\alpha$. If α is a limit ordinal, $G(\omega^\alpha,\omega^\alpha)\leq\sum_{\beta<\alpha}G(\omega^\beta,\omega^\beta)$ since that sum simply repeats some intervals in the construction of $G(\omega^\alpha,\omega^\alpha)$. But if $c\in\gamma$ is a successor and $b\in\beta$ is a successor, then (c,b) is a successor in $G(\gamma,\beta)$, and, more generally, if c is not in the ϵ-th Cantor-Bendixon derivative of γ, and b is not in the δ-th Cantor-Bendixon derivative of β, then (c,b) is not in the $\max(\epsilon,\delta)$-th derivative of $G(\gamma,\beta)$. Hence, $G(\alpha,\alpha)\leq\omega^\alpha$. □

We will define `Push` and `Pop` on an ordinal register called `Stack` which stores the decreasing sequence of ordinals $\beta>\beta_1\ldots\beta_{n-1}\geq\beta_n$, where the last two values are allowed to be equal only if β_n is a limit. The elements of this sequence code formulas. The formula coded by β_1 is being considered, to determine whether it witnesses the truth of β. Each β_{i+1} was found while searching for a witness to the truth of β_i, so the sequence is decreasing.

Definition 12 *We code a finite, monotonically decreasing sequence of ordinals $\beta>\beta_1\ldots\beta_{n-1}\geq\beta_n$, where $\beta_{n-1}\geq\beta_n$ occurs only if β_n is a limit, as* `Stack` $=2^{\beta+1}+\sum_{i=1\ldots n-1}2^{\beta_i+1}+2^{\beta_n}$ *if β_n is a limit, and as* `Stack` $=2^{\beta+1}+\sum_{i=1\ldots n-1}2^{\beta_i+1}+2^{\beta_n+1}$ *if β_n is not a limit.*

We include β_i on the stack as 2^{β_i+1} so that the stack has as its least term the value 2^λ, for λ is a limit ordinal, if and only if that term has been reached as

the limit of considering all finite sequences of ordinals $< \lambda$. That is, whenever we `Push` an element β_i onto the stack, the intended exponent is a successor. A final exponent which is a limit only occurs "magically" at a limit time, and indicates that our infinitely-long attempt to prove the formula coded λ is true has failed. So, we conclude λ is false, and go on.

Recall that ordinal multiplication and exponentiation are defined to be continuous in their second term: $\alpha \times (\beta + 1) = \alpha \times \beta + \alpha$, and for X a set of ordinals, $\alpha \times \sup X = \sup\{\alpha \times x : x \in X\}$. (Lemma 14 about `Push` uses this) $2^{\beta+1} = 2^\beta \times 2$ and for X a set of ordinals, $2^{\sup X} = \sup\{2^x : x \in X\}$. On the other hand, 2^β is isomorphic to the set of finite descending sequences of ordinals less that β, ordered lexicographically:

Lemma 13 *$(\beta_i : i < n) \mapsto \sum_{i<n} 2^{\beta_i}$ is an isomorphism between the set of finite, descending sequences of ordinals all less than β, and 2^β.*

Proof: We construct the inverse: Given an ordinal $\alpha < \beta$, let β_0 be the supremum of those γ such that $2^\gamma \leq \alpha$. Ordinal exponentiation is continuous, so $2^{\beta_0} \leq \alpha < \beta$. If $\alpha \neq 2^{\beta_0}$, let α_1 be such that $\alpha = 2^{\beta_0} + \alpha_1$. Let $\beta_1 = \sup\{\gamma : 2^\gamma \leq \alpha_1\}$. Again, $\alpha_1 \geq 2^{\beta_1}$. If $\beta_1 \geq \beta_0$, then $\alpha \geq 2^{\beta_0} + 2^{\beta_1} \geq 2^{\beta_0+1}$, contradicting the definition of β_0. So $\beta_1 < \beta$. So continue, to find $\beta_0 > \beta_1 > ... > \beta_n$, such that $\alpha_n = 2^{\beta_n}$. The sequence is finite since β is a wellorder. So the sequence of exponents of α is a finite sequence of ordinals, all $< \beta$. Since $\{\gamma : 2^\gamma \leq \sum_{i<n} 2^{\beta_i}\} = \{\gamma : \gamma \leq \beta_0\}$, we have inverted the summation of a decreasing sequence of ordinals. □

Definition 14 *The program* `Push(Stack,` β`)` *is the following routine:*

```
MON Stack;

SCR i, γ = 0, δ;

for δ from 0 to Stack (

   for (i from 0 to 2^(β+1)) (γ++);

   if (γ > Stack) (for Stack to γ; halt)

)
```

Lemma 15 `Push(Stack,` β_i`)` *increases the Stack to the next full multiple of 2^{β_i+1}.*

Proof: So `Push` sets `Stack` equal to $2^{\beta+1} \times \delta$, for δ the least ordinal for which $2^{\beta+1} \times \delta >$ `Stack`. □

Recall how we read the register `Stack` from definition 12. If β was on the stack ($\texttt{Stack} = \sigma + 2^{\beta+1} + \tau$), then `Push` increases τ to $2^{\beta+1}$, leaving `Stack` $= \sigma + 2^{\beta+2}$). I.e., the least element on the stack is changed from β to $\beta+1$. If β was not on the stack ($\texttt{Stack} = \sigma + \tau, \sigma = 2^{\beta_i+2} \times \delta$, for some δ, and $\tau < 2^{\beta+1}$), then `Push` increases τ to 2^{β_i+1}, leaving ($\texttt{Stack} = \sigma + 2^{\beta+1}$). Pushing β onto a stack erases all stack values less than β.

From $\texttt{Stack} = \sum 2^{\alpha_i}$ we will want to read the least exponent α_i which is $\geq$ a certain threshold. We set a "small stack" to be $\tau = \sum_{i>j} 2^{\alpha_i}$, represent the stack as $\sigma + 2^{\epsilon} + \tau$, and check whether $\epsilon = \alpha_i$ is $>$ than the threshold. Unless α is a limit, we will interpret α as a stack element. If α is a limit, we will only be interested in it, in case it is the predecessor of the next-larger exponent, α', in which case α witnesses that α' is false. We can find the representation $\texttt{Stack} = \sigma + 2^{\epsilon} + \tau$ in many ways, but simply trying all possibilities is one option:

We define two functions `Pop`, one to take the smallest value off the stack, and one to take successive values off the stack:

Definition 16 `PopLeast(Stack,` β`)` *is the following routine:*

```
CONSTANT Stack, β;

SCR σ, ϵ;

for ϵ from 0 to β + 1 (

   for σ from 0 to Stack (

      if (σ + 2^ϵ = Stack) ( return ϵ)

   )

)
```

Definition 17 `PopNext(Stack, Threshold,` β`)` *is the following routine:*

```
CONSTANT Stack, Threshold, β;

SCR SmallStack = 0, TempStack = 0, σ, ϵ, κ;

for ϵ to β (

   for σ from 0 to Stack (

      if (σ + 2^(ϵ+1)+SmallStack = Stack) (
```

```
            if (ϵ ≥ Threshold) (return ϵ);

            for TempStack to Smallstack ();

            for Smallstack to 2^{ϵ+1} ();

            for κ from 0 to TempStack (Smallstack++)

        )

    )

);

return ϵ
```

We intend these programs to be applied when the value of `Stack` is between $2^{\beta+1}$ and $2^{\beta+2}$. In that situation, there is always something on the stack smaller than β. If there were not, then these programs would return nothing, which is reasonable when `PopNext` is designed to find a stack element less than β and larger than a given threshold.

Neither program `Pop` really changes the stack. They just read the least element $\beta_j + 1$ of the stack which is not larger than β, or the least element which is `>Threshold`. We read the whole exponent, $\beta_j + 1$, not just β_j, since the stack might contain, as its last term, 2^λ for λ a limit.

Abstract ordinal computers as in definition 9 can compute syntactic operations on the codes of formulas, in the signature Σ. We would like to show that ordinal computers can compute, in addition to Σ, the truth predicate T, determining whether any Δ_0 formula is true. Let's gather all of the syntactic formula manipulation into an abstract ordinal program called W for Witnessing, as is done in [6], section 6. We have used this notion already in lemma 9.

Definition 18 *Let $W(\beta, \gamma, T(\gamma))$ be an abstract ordinal computer program that determines whether γ and its truth value $T(\gamma)$ are sufficient information to witness the truth of β. Let the output of W be 0 unless some pair $(\gamma, T(\gamma))$ with $\gamma < \beta$ witnesses the truth of β, in which case W outputs 1. In particular, $W(\beta, \gamma, T(\gamma))$ is the program which finds the syntactic structure of β, and then*

- *if β codes an atomic sentence with constant symbols for ordinals and for $T(\gamma)$, in the signature $\{<, G\}$, the program W evaluates that atomic sentence.*
- *if β and γ code the sentences ϕ and $\neg\phi$, $W = 1 - T(\gamma)$.*
- *if β codes the sentences $\phi \vee \psi$ and γ codes ψ, then $W(\beta, \gamma, T(\gamma)) = T(\gamma)$.*
- *if β codes the sentences $\exists x < c\phi$, where c is a constant symbol, and if γ codes the sentence $c' < c \wedge \phi(c'/x)$ where the constant c' replaces the variable*

x, then $W(\beta, \gamma, T(\gamma)) = T(\gamma)$.

Then β is true iff there is some witness $\gamma < \beta$ such that $W(\beta, \gamma, T(\gamma)) = 1$. We will find the truth value of β by searching through decreasing sequences of ordinals $< \beta$, until we find a witnessing sequence, a stack which conveys its witnessing – through pairs of ordinals of the form $\alpha' > \alpha$ such that $W(\alpha', \alpha, T(\alpha))$ holds, or of the form $\alpha' = \alpha + 1$ such that $T(\alpha)$ is known – from a limit ordinal α which appears twice in the stack. This situation arises as the limit of a search over all ordinal sequences $< \alpha$ during which we did not find a witnessing sequence for α. This is the falsehood from which we conclude, via the witnessing sequence, the truth value of β.

Definition 19 *Say* $\beta = \beta_0 > \beta_1 > \beta_2 ... > \beta_{n-1} = \beta_n$ *is a witnessing sequence for* β *if for each* $i = 1 \ldots n-1$, $W(\beta_{i-1}, \beta_i, T(\beta_i)) = 1 = T(\beta_{i-1})$ *or* $\beta_{i-1} = \beta_i + 1$ *and* $T(\beta_{i-1}) = 0$ *and* β_n *is a limit ordinal and* $T(\beta_n) = 0$.

The terminal value of the stack will be $2^{\beta+1} + \sum_{i=1 \ldots n-1} 2^{\beta_i+1} + 2^{\beta_n}$, where $\beta_n = \beta_{n-1}$ is a limit, and no other β_i is a limit. That last summand witnesses that we have examined every possible witness for β_{n-1} and found none, hence β_{n-1} is false. Each summand then witnesses the truth value of the preceding summand, back to β, and we are done. We cannot simply loop through *all* decreasing sequences. If we know that β_j is true, but that β_j doesn't witness β_{j-1}, we must skip the sequence $\cdots \beta_j, \beta_j - 1 \cdots$, since that sequence, as soon as we know the truth value $T(\beta_j - 1)$ and check that $W(\beta_j, \beta_j - 1, T(\beta_j - 1)) = 0$, we intend to interpret to mean that β_j is false. We should only reach that sequence if no $\beta_{j+1} < \beta_j$ could witness that β_j is true. This "skip" is performed by `Push`-ing the Stack to $\sum_{i<j} 2^{\beta_i+1} + 2^{\beta_j+1} + 2^{\beta_j+1}$. Of course, this also speeds up the program: once we know that β_j is true but that that our current witnessing sequence $2^{\beta+1} + \sum_{i=1 \ldots j-1} 2^{\beta_i+1} + 2^{\beta_j+1}$ doesn't witness β's truth, we move on, and consider $2^{\beta+1} + \sum_{i=1 \ldots j-1} 2^{\beta_i+1} + 2^{\beta_j+2}$.

Definition 20 `Truth(`β`)` *is the following program:*

```
CONSTANT: β;

MONOTONE: Stack, i;

SCRATCH: α, α';

Push(Stack,β);

for i from 0 to 2^β (

   α = PopLeast(Stack, β);

   if α is a successor (Stack ++; α = 0);
```

```
α′ = PopNext(Stack, α, β);
if α′ ≠ α (Push(Stack, α));
if α′ = α (
ν = 0; % This is the truth value of α′.
   while α′ ≤ β (
      if α′ = β (return ν);
      α = α′;
      α′ = PopNext(Stack,α + 1,β);
      if W(α′,α,ν)= 0 and α′ ≠ α + 1 (
         α′ = β; % to terminate the while loop
         Push(Stack,α)
      );
      if W(α′,α,ν)= 1 (ν = 1);
      if W(α′,α,ν)= 0 and α′ = α + 1 (ν = 0)
   )
)
)
```

If there is a witnessing sequence for β, then this search will find it. The only stack value which witnesses β being false is $2^{\beta+1} + 2^{\beta}$ if β is a limit, and if β is a successor, then $2^{\beta+1} + 2^{\beta} + \tau$, where τ witnesses the truth value $T(\beta - 1)$, and $W(\beta, \beta - 1, T(\beta - 1)) = 0$.

Theorem 21 `Truth(`β`)` *computes the truth value of the sentence in the language* $\{\in\}$ *with constant parameters which* β *codes.*

Proof: We reduce truth in ZFC with parameters to a computation of the recursive truth predicate for the constructible universe, as in ([6], section 6). Then we write an abstract ordinal program to compute the syntactic operations, as in lemma 10, to satisfy definition 18. As we explained before and after definition 17, a proof of $T(\beta)$ is contained in a witnessing sequence $\beta > \beta_1 > \cdots beta_i > \cdots \beta_{n-1} = \beta_n$. If `Stack` codes a witnessing

sequence with the coding described in definition 11, then `Truth(`β`)` will halt and return the truth value of β, for in the computation of `Truth(`β`)`, the pair $(\alpha' + 1, \alpha)$ become the least two exponents of the stack. If α is a limit and $\alpha' = \alpha$, then the while loop repeatedly sets $(\alpha'\alpha)$ equal to each pair (β_i, β_{i+1}) of stack elements and checks that $T(\beta_i) = 1 = W(\beta_i, \beta_{i+1}, T(\beta_i))$ or $T(\beta_i) = 0 = W(\beta_i, \beta_{i+1}, T(\beta_i))$ and $\beta_i = \beta_{i+1} + 1$. We need to know that `Stack` will eventually code the witnessing sequence for β. But focusing on how `Push(Stack,` $\cdots$`)` is called in the program, we see that `Stack` will eventually code every decreasing sequence of ordinals $\beta > \beta_1 > \cdots beta_i > \cdots \beta_{n-1} \geq \beta_n$ for which $\beta_{i+1} \leq$ the least witness for $T(\beta_i)$. $\square$.

3 How many registers are necessary in a universal ordinal register machine?

Consider, first, ordinary register machines storing natural numbers.

Definition 22 *A register machine has the following three commands*

- `Zero(`x`)` *: erases the value of register* x*.*
- x `++` *: increments the value of register* x*.*
- `if` $x = y$ `goto` i `else` j *: a general switch.*

A **For** *program uses* `goto` *loops only to model the commands*

- `for` x `from` 0 `to` z `(loop)`*.*
- `if` $(x = y)$ `(instructions)`*.*

A **While** *program lacks* `Zero(`x`)` *and* `goto`*, but has the commands*

- x `--` *: decrements the value of register* x*.*
- `while(`$x > 0$`;` x`--) loop`*.*

Theorem 23 (*[7] p. 205*) 5*-variable* `While`*-programs simulate Turing machines.*

The proof is by storing the bit strings on the Turing tape left and right of the active head as register values. When the active head goes right, the bit string to the left increases by $2\times$, and the bit string to the right decreases by $1/2$.

Theorem 24 (*[7] pp. 255-8*) `While`*-programs using* 2 *variables can simulate all* `While`*-programs.* `FOR`*-programs using* 3 *variables can simulate all* `While`*-programs.*

The proof is by storing all the registers as $2^{x_0} \times 3^{x_1} \times ...{p_n}^{x_n}$, then copying these values to another register, and meanwhile altering or comparing them according to how the many-register program would have altered or compared them in its active command.

Definition 25 *Let OC^n be the set of n-register well-structured ordinal computer programs (as in definition 2). Say $\rho : Ord^n \to \alpha^n$ reflects OC^n if for each P in OC^n, the function f_P which takes the inputs to P to the output of P, commutes with ρ. Let L_n be the vocabulary with a function for each n-register program: $L_n = \{Ord, <, =\} \cup \{f_P : P \in OC^n\}$, and let $FO^k(L)$ be the first order formulas in the language L, to quantifier depth k.*

Definition 26 *Let ρ_0 be the function $\rho_0(\alpha) = \alpha \mathrm{mod} \omega$.*

Let ρ_1 be the identity below ω, and be $\omega + \rho_0$ above ω.

Let ρ_2 be the identity below $\omega \times 2$, and be $\omega \times 2 + \rho_0$ above $\omega \times 2$.

Let $\rho_3(\alpha) = \alpha \mathrm{mod} \omega^\omega$.

Let ρ_4 be the identity below ω^ω, and be $\omega^\omega + \rho_3$ above ω^ω.

Let $\rho_5(\alpha, \beta)$ be the pair $(\rho_4(\alpha), \rho_4(\alpha) + \rho_4(\beta - \alpha))$ if $\alpha \leq \beta$ and be undefined if $\alpha > \beta$.

Lemma 27 *$\rho_1 : Ord \to \omega \times 2$ reflects OC^1, is the minimal reflection preserving $FO^1(L_1)$, and preserves even $FO^2(L_1)$. ρ_2 preserves $FO^3(L_1)$.*

Proof: In a well-structured program with only 1 variable, `for` a `to` a `(`L`)` never executes its loop, and `if` $a = a$ `(`L`)` always executes its instructions. The result of the computation, on input a, is $a + n_P$ or n_P, depending on whether the instruction `Zero`(a) occurs and executes. It is easy to check that $\rho(P(a)) = P(\rho(a))$. If $\forall a P(a) \neq Q(a)$, then, as P and Q are constants or linear functions, we get four cases, in all of which $\forall a \rho(P(a)) \neq \rho(Q(a))$, and similarly for $\forall a P(a) < Q(a)$ and other atomic relationships replacing $\neq$, we can check the language's preservation. □

Lemma 28 *$\rho_5 : Ord^2 \to (\omega^\omega \times 3)^2$ reflects OC^2, is minimal such that it preserves $FO^2(L_2)$, and preserves $FO(L_2)$.*

Proof (that ρ_5 is minimal): If $L_{\times\omega}$ `=` `Zero`(b)`;` `for`b `to` a `(`$a++$`).`, then $L_{\times\omega}(a, b) = (a \times \omega, a \times \omega)$. (Proof: Let a initially be a_0. When b reaches $a_0 \times n$, a reaches $a_0 \times (n + 1)$. □) If $L_{\times\omega^n} = L_{\times\omega}$ repeated n times, then $L_{\times\omega^n}(a, b) = (a \times \omega^n, a \times \omega^n)$; If L_p `=` `Zero`(b)`;` `for` b `to` a `(for` b `to` a `(`$a++$`);` $a++$, then $L_p(a, b) = (a \times \omega + \omega^2, a \times \omega + \omega^2)$. (Proof: The first run through the inner loop produces $(a \times \omega + 1, a \times \omega)$. Further runs through

the inner loop produce $(a \times \omega + \omega \times n + 1, a \times \omega + \omega \times n)$, which are finally equal at $(a \times \omega + \omega^2, a \times \omega + \omega^2)$. In this way, we can generate L_q for any linear (in a) polynomial (in ω) $q(a, \omega)$ we wish to see as the output. □ Proof (reflection) First, observe that $P \in OC^2$ is equivalent to a program $P' \in OC^2$ which is only one loop deep.

For instance, we can write a two-loop-deep program to produce the value ω^2: `Zero(`x`); for` x `to` y `(for` x `to` y `(`y`++); `y`++)` takes any finite input to ω^2, just the same as running y up to ω and then running x up to y. Similarly, `Zero` x`; for` x `to` y `(for` x `to` y`(for` x `to` y`(`y`++); `y`++); `y`++)` takes any finite input to ω^3, just the same limit as running y up to ω, then running x up to y, then running y up to x. The proof relies on the rule in definition 2 which prevents a loop index or bound from being erased. As a result, the order between them is fixed, and can only be made to fail during the loop by incrementing the index. Then, this finite difference can be exploited by an interior `for` loop. However, the variables could be imagined to be switched, then, so that the order relation "index < bound" can be imagined to be strict throughout the whole operation of the main loop. In this case, repeatedly chasing the bound only results in finding the next "limit of f-closed ordinals," and ω^n provide infinitely many limits of limits of... f-closed ordinals, where f is any function that can be produced within an interior loop. Those same functions can be computed, then by a sequence of loops without inner loops, which push the loop bound high enough, and then run the index up to it.

As was observed in the run of L_p now happens generally: after the inner loop has run, $a = b$. Subsequent operations inside the outer loop can only make b finitely larger than a. Second, inside any loop, the loop index grows at least linearly in time, and the loop bound grows at most linearly in time. To "Zero" the index or bound of a loop, in the loop, is illegal by definition 2, so if a loop is called, the order relation between the variables is fixed (up to a finite amount) throughout. An interior `for` loop forces the loop and index to be the same, and `if` has no effect on the values. So for any $P \in OC^2$, P is bound by a function $q(a, \omega)$, linear (in a), polynomial (in ω). □.

Lemma 29 *Ord^3 reflects below $\epsilon_{\omega \times 4}$, and not lower.*

Proof (not lower): The program $y++$`; for` x `to` y `(Zero(`z`); for` z `to` y `(`$y++$`))` halts at the first ϵ-number (closed under $\alpha \to \omega^\alpha$) above the initial value of y. Repeating the loop n-many times finds the n-th ϵ-number above the initial value of y. □ Proof(reflection): Suppose the first loop is $L_0 =$ `for(`$x = a; x < y; x++$`),` where a can be x, z, or 0 (same as `Zero(`x`); for(`$x = x$`...)`. This same loop format can be repeated, as in `for(`$x = 0; x < y; x++$`) (for(`$x = x; x < y; x++$`) (`L`); `$y++$`)`. Let $f(x, y, z)$ be the supremum of the register values after applying the loop L to the initial register values x, y, z. The inner loop ends when x reaches an ordinal γ which

is closed under f (γ is f-closed if $f(x,y,z) < \gamma$ whenever $x,y,z < \gamma$). Then in the outer loop, y increments, and so we reach γ_1 f-closed, and so on. The outer loop ends at the first γ which is a limit of f-closed ordinals. Now $x < y$ is fixed for the duration of the computation. For if x were incremented above y infinitely often, then y is also incremented above x infinitely often (before each consideration of the bounding clause $x < y$), so that at time $\sup t_i$, where t_{2i+1} is the next time x is larger, and t_{2i+2} is the next time y is larger, then $x = y$ again, and as this is a limit time, we are checking the bounding clause $x < y$, and the loop ends. So if x exceeds y infinitely often, then the loop ends. So, without loss of generality, suppose $x < y$ always holds, and consider what an inner loop can do. Incrementing x is counter-productive, since it hastens the time when $x = y$ will be attained. Incrementing y is a great idea, but the only clock available is `Zero(z); for z to y(f(y))`, which executes until $z + \alpha$, incrementing once each loop, reaches $f^\alpha(y)$, the α-th iteration of f, whatever function is in the innermost part. This function could, at most, be `for z to` x or `for x to` z, in which cases $f(y)$ would increase y some infinite number of times, but never more than its own value, so $f(y) < y + y$.□

If the initial register values are 0, then we cannot compute anything beyond ϵ_ω. But if the initial register values are given, then we reflect the first one into $(\epsilon_\omega, \epsilon_{\omega\times 2})$ and the next into $(\epsilon_{\omega\times 2}, \epsilon_{\omega\times 3})$, the third into $(\epsilon_{\omega\times 3}, \epsilon_{\omega\times 4})$, and the same proof shows that subsequent computation stays below $\epsilon_{\omega\times 4}$.

Theorem 30 *An ordinal computer with fewer than four registers cannot be a universal ordinal computer. However, an ordinal computer with ten registers can model a universal ordinal computer.*

Proof: We have proven in the lemmas that fewer than four registers is insufficient, since these computers reflect below small ordinals. The program in definition 20 is written using ten registers. That is, it uses the five variables β, `Stack`, i, α, α', (we can recompute the loop limit 2^β each time we check $i < 2^\beta$) and then calls `Pop`, which uses as local variables a `Small Stack`, a `Temp Stack`, ϵ to search between 0 and $\beta + 1$, γ, and a fifth register, which might sometimes store the sum $\alpha + 2^\epsilon +$ `SmallStack`, and sometimes be the loop index κ in the last line of `Pop`. The register for `Pop`'s γ, which we could call `Large Stack` in analogy with `Small Stack` never exceeds `Stack`; we can make it larger than `Stack` when it's time for the `while` loop in definition 20 to halt. If γ can code the bit of information that halts the `while` loop, then that loop doesn't need a register dedicated to indexing it. □

We would like to indicate how four registers are sufficient for a universal program on an ordinal register machine. We simulate an n-register machine by putting all n variables onto two stacks. We copy the information from one stack to the other, and change the appropriate i-th register in the process, as in the proof of theorem 24. The fourth variable contains the value of a single

element. When that element is erased on the stack, we copy its value more deeply into the stack, where it won't be erased by the varying and limiting of values lower on the stack. We do not have a clear and convincing proof of this.

Conjecture 31 *Four registers suffice for a universal program on an ordinal register machine.*

4 Complexity

For ordinal register machines, it is possible to compare the runtime of a program to its input values, and therefore it is reasonable to talk about the bounds on the complexity of problems for such machines.

Our program for computing truth (definition 20) runs in time at most ordinal-exponential in the input β. A similar program, described in [5], runs in ordinal-polynomial time: to determine the truth predicate, when ordinally many bit registers are available, search the registers below α to find a witness for α. This takes time $\sum\{\beta : \beta < \alpha\}$ If $\alpha = \omega^\gamma$ for some γ, this is α. In any case, the sum is $< \alpha^2$. This program runs faster than definition 20 because it can store the whole recursive truth predicate up to β when computing $F(\beta)$. It seems intuitively clear that a computer with finitely many ordinal registers cannot run in time faster than $O(2^\beta)$, i.e., that it must compute $F(\beta_1)...F(\beta_n)$ for every finite sequence $\{\beta_i : i < n\}$ of ordinals $< \beta$.

References

[1] R. Bissell-Siders, *Ordinal computers.* math.LO/9804076 at arXiv.org, 1998.

[2] H. Friedman, *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory.* Logic Colloquium '69 (Proc. Summer School and Colloq., Manchester, 1969), pp. 361–389. North-Holland, Amsterdam, 1971.

[3] G. Jacopini and C. Böhm, *Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules.* Comm ACM, 9,5 May 1966.

[4] J. Hamkins and A. Lewis, *Infinite Time Turing Machines.* J. Symbolic Logic, 65(2): 567-604, 2000.

[5] P. Koepke, *Turing Computations on Ordinals.* Bulletin of Symbolic Logic, 11(3): 377-397, 2005.

[6] P. Koepke and R. Siders, *Register Computations on Ordinals.* submitted Feb 2006. For the moment, see

http://www.math.helsinki.fi/∼rsiders/Papers/RegistersOnOrdinals/

[7] M. Minsky, *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

[8] J. Shepherdson and H. Sturgis, *Computability of recursive functions*, J. Assoc. Comput. Mach. 10 217–255, 1963.

[9] J. Tucker and J. Zucker, *Computable functions and semicomputable sets on many sorted algebras*, in S. Abramsky, D. Gabbay and T Maibaum (eds.) *Handbook of Logic for Computer Science*, Volume V, Oxford University Press, 317-523.